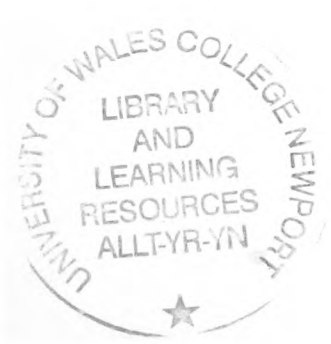


GWENT COLLEGE OF HIGHER EDUCATION

FACULTY OF TECHNOLOGY

THE APPLICATION OF OPTIMAL TRANSPUTER ARCHITECTURE TO CONCURRENT PROCESSING IN THE IMPLEMENTATION OF VISION PROCESSING ALGORITHMS

A thesis submitted to

THE COUNCIL FOR NATIONAL ACADEMIC AWARDS

in Partial Fulfilment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

By

IAN BRAMLEY BENNETT, B.Sc. (Hons), AMIEE

Newport, Gwent, U.K.

MAY 1989

To my parents,

Cliff and Sheila

"You yourself spend highly valuable time and much effort in playing chess. Why do you do it? What do you get out of it?"

"Why, I...uh...mental exercise, I suppose...I like it!"

"Just so. And I am sure that one of your very early philosophers came to the conclusion that a fully competent mind, from a study of one fact or artifact belonging to any given universe, could construct or visualise that universe, from the instant of its creation to its ultimate end?"

"Yes. At least, I have heard the proposition stated, but I have never believed it possible."

"It is not possible simply because no fully competent mind ever has or ever will exist. A mind can become fully competent only by the acquisition of infinite knowledge, which would require infinite time as well as infinite capacity. Our equivalent of your chess, however, is what we call the 'Visualisation of the Cosmic All'."

First Lensman Samms meeting the Arisian 'Mentor'
First Lensman - E.E. 'Doc' Smith
Panther Books, 1973

DECLARATION

It is hereby confirmed that the Author, while registered as a candidate for the degree for which this thesis submission is made, has not been a registered candidate or enrolled student for another award of The Council for National Academic Awards, or other academic or professional institution, during the research programme.

No material contained in this thesis has been used in any other submission for an academic award.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to thank Dr. David W. Downing for his considerable efforts instigating and supporting this work, for his continual interest, encouragement and backing, and for being not only my Mentor and Supervisor, but also a friend. I would also like to thank David for his many helpful comments on drafts of this thesis.

Thanks to the College library staff for their help and assistance, especially Mrs. Pam Cobley. Thanks also to Mr. Graham Wilson for the helpful discussions we have had. I would like to express my appreciation to Mr. Huw Thomas for occasional help given with the College "Red Tape", and to Mr. Edward Tate for occasional help with the laser printer, during the printing of this thesis.

Grateful acknowledgement is given to Gwent College of Higher Education for financing this work, to The Science and Engineering Research Council (SERC) for the occasional use of their transputer facilities at The Rutherford Appleton Laboratory, Didcot, and to Inmos Ltd., Bristol, for donating some non-commercial transputer devices to the project under the SERC Transputer Initiative.

Last but not least, I would like to express my appreciation to Helen, for her patience, understanding, and encouragement. Thank you Helen.

THE APPLICATION OF OPTIMAL TRANSPUTER ARCHITECTURE TO CONCURRENT PROCESSING IN THE IMPLEMENTATION OF VISION PROCESSING ALGORITHMS.

IAN BRAMLEY BENNETT

ABSTRACT

Repetitive low level image processing transformations can be performed at high speeds by SIMD arrays, DSP and dedicated VLSI devices. These strategies cannot be adopted with more complex and time consuming data dependent algorithms. A flexible and programmable component must be used, and the use of many such devices in parallel, using dynamic load balancing techniques, is necessary to enable acceptable execution performance to be obtained.

The transputer is a powerful new microprocessor with unique on chip communications facilities. Together with the new parallel programming language, occam, the transputer was specifically designed for parallel processing applications. Large transputer networks can be used for computationally intensive applications.

This work has investigated the use of transputers for performing image processing algorithms of all three levels of complexity. Techniques were devised and implemented for the execution of low, medium and high levels of image processing algorithms on a multi-transputer network. A software architecture using SUPPLY and DEMAND processes was designed, and dynamic work load balancing was achieved, operating on a ternary tree network of up to 32 transputers.

Some 80 image processing algorithms were successfully implemented within the software architecture. In particular, the more complex operation of Feature Extraction was achieved using the multi-transputer system. The Features extracted, involving Convex Hull, Convex Hull Deficiencies, Areas and Perimeters, and Shape Factors were used to build a Feature Vector. The use of this Feature Vector in Scene Interpretation, to realise Learn and Recognise functions has been investigated.

The results of the work clearly show that while the system proposed is not as effective at executing repetitive, data intensive transformations as methods mentioned earlier, it can execute more complex Feature Extraction and Scene Interpretation algorithms efficiently. An Efficiency of 85% was achieved for Convex Hull formation, using 32 transputers.

KEY TO ABBREVIATIONS

ALU - Arithmetic Logic Unit

CAS - Column Address Strobe

CH - Convex Hull

CHD - Convex Hull Deficiency

C-MOS - Complementary Metal Oxide Semiconductor

DRAM - Dynamic Random Access Memory

DSP - Digital Signal Processor

DMA - Direct Memory Access

EMI - External Memory Interface

EPROM - Erasable Programmable Read Only Memory

FPU - Floating Point Unit

IC - Integrated Circuit

Kbyte - Thousand bytes (2^{10} = 1024)

Mbit - Million Bits

Mbyte - Million bytes ($2^{10} * 2^{10}$ or 2^{20} = 1048576)

MBPS - Million Bits Per Second

MFLOPS - Million FLoating point Operations Per Second

MIMD - Multiple Instruction Multiple Data

MIPS - Million Instructions Per Second

MISD - Multiple Instruction Single Data

N-MOS - N-type Metal Oxide Semiconductor

PB - Partial Boundary

PCB - Printed Circuit Board

PCH - Partial Convex Hull

RAM - Random Access Memory

RAS - Row Address Strobe

RISC - Reduced Instruction Set Computer

ROM - Read Only Memory

SERC - Scientific and Engineering Research Council

SIMD - Single Instruction Multiple Data

SISD - SIngle Instruction Single Data

SRAM - Static Random Access Memory

TDS - Transputer Development System

TTL - Transistor Transistor Logic

VDU - Visual Display Unit

VLSI - Very Large Scale Integration

WSI - Wafer Scale Integration

Folds

```
...   Fold, inside this structure are lines of occam.


{{{  Start of opened fold

Lines of occam inside fold.

...   Another fold inside the outer fold.

}}}  End of opened fold
```

CHAPTER 1

INTRODUCTION

1.1 Research Project Introduction

Extensive work has been featured in the literature on the implementation of low level image processing algorithms using dedicated VLSI components [1, 2, 3, 4], parallel SIMD arrays of simple processors [5, 6], and transputer arrays [7, 8]. Processor arrays can be very effective for the non data dependency and simple operation of these transformations, because the amount of computation to be performed is replicated over the entire image. Very high speed solutions have been realised, allowing image enhancement and other simple operations to be performed in real time. Very little has been published, however, on the parallel implementation of higher level image processing algorithms, and more work must be done to understand this better.

In any real application, such as automated industrial inspection, a combination of low, medium and high level image processing algorithms are essential in order to achieve the desired objective [9]. As more complex algorithms will generally require considerably more computational resources and time than less complex ones, especially if the latter are performed using VLSI or SIMD arrays, it would be useful if the performance of the complex algorithms could also be enhanced.

The objectives of this research project were to implement standard useful image processing algorithms of different levels of complexity using a multi-transputer system. In particular, medium and high level image processing operations were to be studied, and implemented on the system. This should result in more efficient and faster versions of the more complex algorithms, reducing the overall execution times for practical image processing applications.

Hardware had to be designed, and software devised, so that medium and high level image processing algorithms could be implemented in parallel. A requirement of the software was the facility for sharing the processing workload between the available processors, to enable a significant speed up advantage over that obtained by a single processor.

1.2 Introduction to Parallel Processing

Technological Developments. The technology of Integrated Circuit (IC) manufacture is improving rapidly. Each year more logic gates, which make up any digital memory or microprocessor device, can be fabricated on one die, by increasing the density. At the time of writing, 4 Million bit (Mbit) memory devices are being tested in the laboratory, and only a short time previously 1 Mbit devices were in the same stage of development. Complex microprocessor circuits are now being produced with over 350,000 logic gates, resulting in extremely complicated and powerful components [10]. As logic circuits are made smaller, with reduced propagation delays, they can be driven at higher speeds, resulting in faster components. It has been observed that projecting the current

technological developments into the 1990s, ever more functionality will be squeezed onto a single IC, resulting in processors being smaller, faster, and more powerful [11].

A specific example of these trends can be illustrated by considering transputer devices. The processor cycle time will be reduced from 50 nS to 20 nS (50 Mhz clock), the amount of on-chip memory could be increased from 4 Kbytes to 32 or 64 Kbytes, with access times about 2-4 times faster, more higher speed communication links could be incorporated, and more functions currently implemented in microcode could be executed by dedicated silicon. All these enhancements will result in the transputer processor being much more powerful than at present [12].

The Need for Parallel Processing. These technological projections, however, do not satisfy the demands being put upon microprocessors by computationally intensive applications such as computer vision, expert systems, and intelligent machines. Using a conventional sequential processor, each stage in the task is performed sequentially, until the overall result is achieved. The execution of such applications can only be improved by increasing the speed of the processing device or changing the algorithm of the task being performed. If it is assumed that the algorithm is already optimal, then execution improvements can only be obtained by changing the processing device. To achieve this, a faster equivalent may be substituted, utilising more advanced technology. Even using higher speed Gallium Arsenide (GaAs) devices, these solutions only yield speed improvements of an order or two, which is insufficient.

Alternatively, the fundamental system of processing may be changed, to utilise dedicated hardware, or multi-processor parallel processing. In most applications, sections of the computation could be executed simultaneously, in parallel. Parallel processing would allow the overall task to be achieved in a dramatically shorter time, because many of the sections would be executed concurrently, and not sequentially, as before. Only by using parallel processing techniques and technologies can the computational demands of advanced complex applications be met.

Parallel Processing Paradigms. There are three important hardware paradigms for parallel processing - pipelines, farms, and arrays [12]. There are also essentially two software paradigms found in parallel processing applications - Algorithmic and Geometric parallelism [13].

Algorithmic Parallelism on a Pipeline. Pipelines are one of the simplest multi-processor configurations in terms of connectivity, but can require quite extensive design effort in order to reduce the impact of bottlenecks in the data / computation path [7] (figure 1.1).

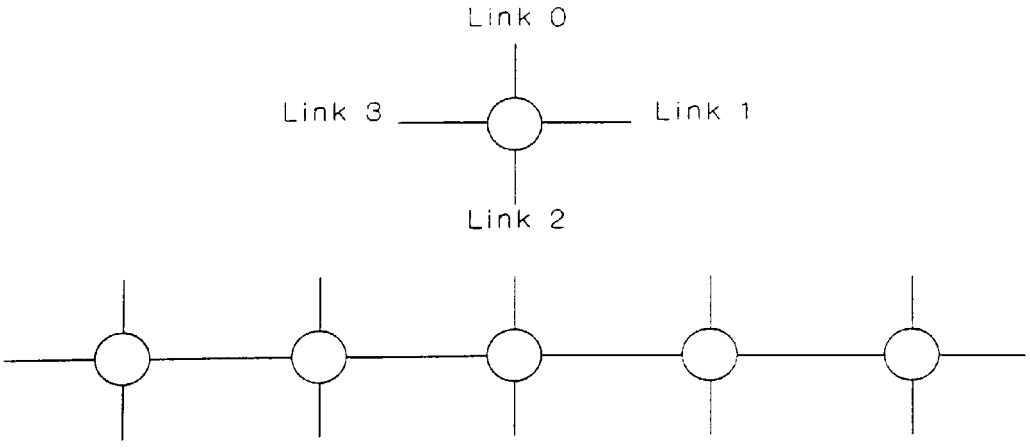


Figure 1.1. Simple Node and Pipeline Configuration.

Algorithmic parallelism typically involves a sequence of operations that are repeatedly applied to streams of data. Such operations may be mapped onto individual processors to increase the speed of the whole sequence. The main advantage with pipelines used in this way is that, if there were as many processor stages as there were algorithm stages, a good one to one mapping would be achieved, yielding a good speed up with the minimum of implementational effort.

Pipelines can be most useful when dealing with streams or sequences of data, which are required to be transformed continually using low level, non data dependent image processing algorithms. The input data is presented to the first stage, and when stage 1 processing is complete the transformed data is passed to the second stage. The second stage outputs the data to the third stage when it has completed its computation, and so on, to the end of the stage. There are several disadvantages to this arrangement however.

The latency between the initial data being presented to the first stage, and transformed data being output from the *nth* stage, is

$$\text{latency} = \sum_{1}^{n} t_{(i)} \qquad \text{where } t_{(i)} \text{ is the time taken by stage i.}$$

$$i = 1,2,3 \ldots n$$

This means that no data will be output from the system until the first set of data has been processed by all the stages, and passed through the system.

When the first batch of data has been output from the nth stage of the pipeline, the next data will be output time

t_{max} later, where t_{max} is the time of the longest stage operation of the pipeline. This can represent a bottleneck, limiting the overall throughput, if t_{max} is significantly larger than other stage times in the pipeline. Significant design effort must be applied to ensure that bottlenecks are kept to a minimum [7]. It is also important to ensure that t_{max} is less than the period of the data.

It is not straightforward to increase the performance of such a system. The algorithm stage with the longest processing time, t_{max}, must be divided between several processors, executing concurrently, in order to reduce the latency of the throughput.

It would be difficult to map an arbitrary number of image processing algorithms, designed for a certain application, on to processors in a pipeline. Different image processing tasks require different numbers of image processing operations applied to a data set in a predetermined sequence. Some algorithms will also require significantly more processing time than others.

A simple pipeline is extendible in that extra processors can be easily added at the end, if required, to increase the system performance. Pipelines involving Algorithmic parallelism, however, may limit this to a fixed number of processors.

Processor Farming on a Pipeline. A pipeline can also be used for implementing a processor farm [14]. The software required for the data and communications routing is more straightforward than with some other configurations. Usually a single channel pair is used for communications up

and down the pipeline, although this may be more complicated to offer a degree of fault tolerance, as shown in figure 1.2. This approach yields a simple processor farm implementation without extensive software.

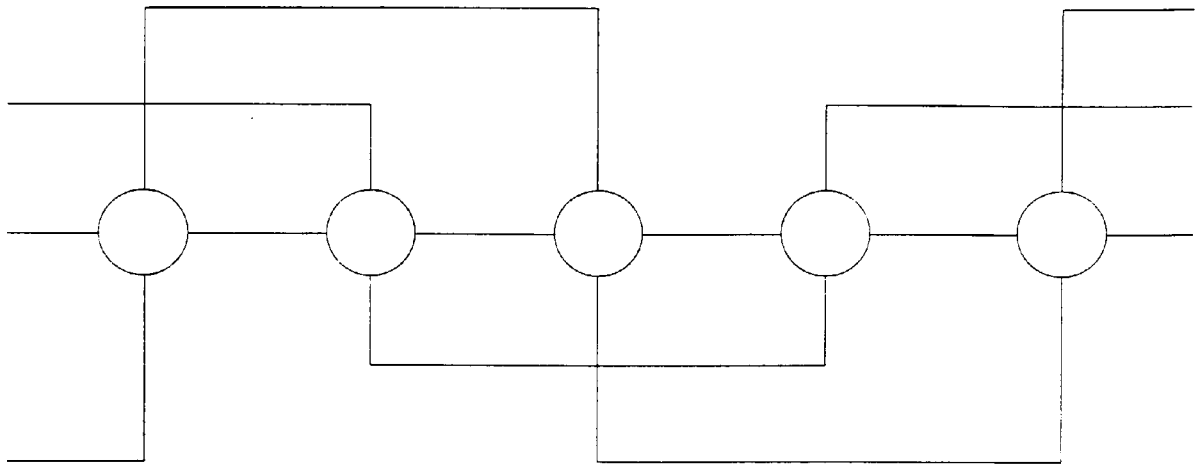


Figure 1.2. Fault Tolerant Pipeline.

The main problem with using a pipeline for a processor farm, especially with image processing, is that input and transformed data must be passed up and down the length of the pipeline. This can lead to the pipeline becoming communication bound, particularly with data intensive low level image processing.

Processor Arrays. Processor arrays have been considered and used for a variety of image processing operations [15, 16, 8, 17] (figure 1.3).

SIMD configurations have been proposed in arrays, cubes, pyramids, and hypercubes. Typically there are *N* processors and *N* memories. A control unit broadcasts instructions to all the processors, and each execute the same instruction simultaneously. The processor interconnection network allows local data communications to take place directly.

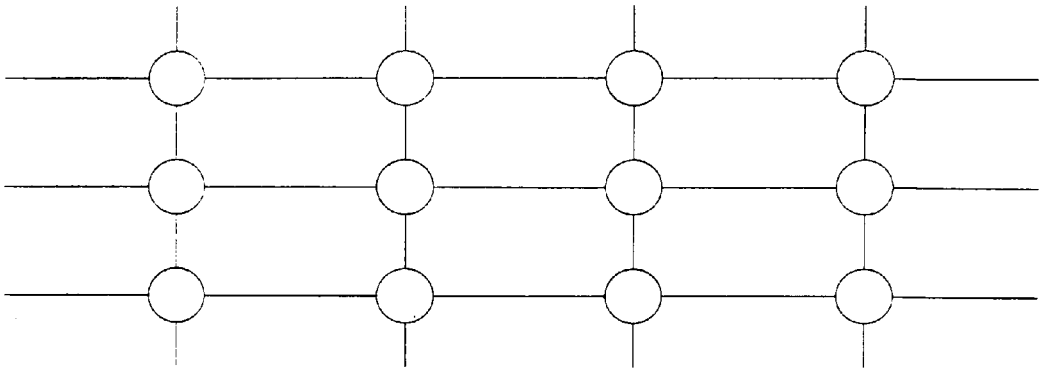


Figure 1.3. Regular Array Configuration.

Arrays of simple processor devices may be configured to work systolically, where image data is passed through the array continually, transformed, and passed out. The data may also be passed back again, for further transformations. This type of approach is often employed where repetitive transformations are continually required on different data sets.

Many low level non data dependent image processing operations exhibit geometric parallelism [18, 9], which can make the implementation of parallel image processing efficient and straightforward. Each processor works on that part of the image corresponding to its own position in the array. Due to the non data dependent nature of these algorithms, all processors are typically computationally evenly weighted.

The values of neighbouring pixels that are not contained on a processor may be obtained from the appropriate neighbouring device. Typically edge pixels such as these are exchanged between processors after operations have been performed.

As the size of the array grows, however, the time taken for

instructions to be broadcast throughout the array increases [12].

Hypercubes. A more complicated configuration, termed Hypercubes, can be designed with various dimensions. Figure 1.4 illustrates some common arrangements. Not all of these Hypercubes can be realised using standard transputers, however, as dimensions of 3 and above require more than four communication links. These extra links could be provided by memory mapping transputer link adapters, however, or by combining two or more transputers into a single node, as illustrated in figure 1.5.

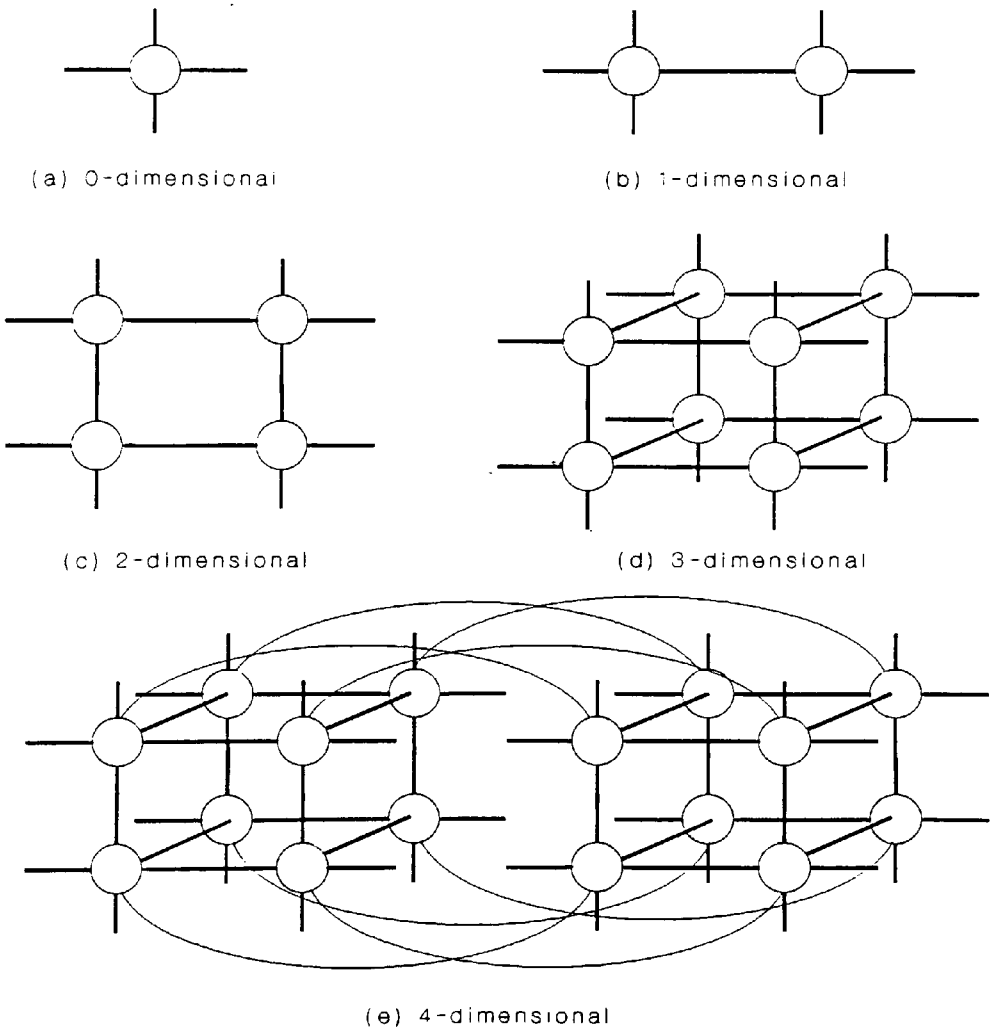


Figure 1.4. Hypercubes of Various Dimensions.

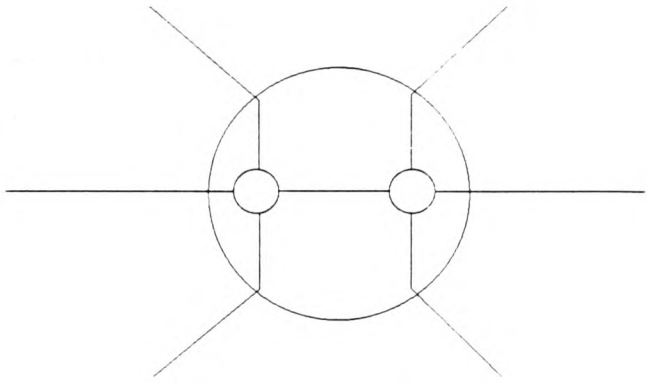


Figure 1.5. Combining Two Transputers into a Single Node.

Ideally Connected Structure. Ideally connected structures, (figure 1.6) may be designed, where each node is connected to every other node. This arrangement would ensure optimal connectivity, but as with Hypercubes this configuration requires more than four communication links per node for structures with more than five nodes. The structure using five nodes does not have any spare nodes for external connections, although a simple two transputer node would overcome this limitation, as mentioned above.

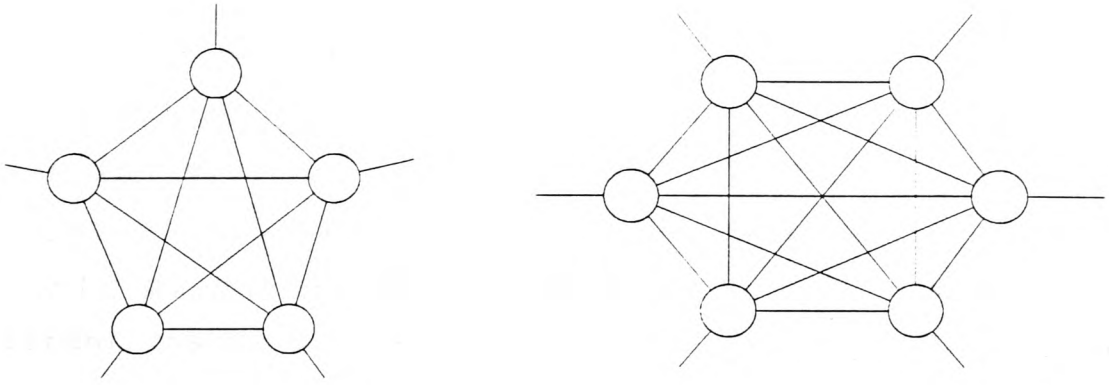


Figure 1.6. Ideally Connected Structures.

Types of Parallel Processing Hardware. Several types of parallel processing have been featured in the literature. One classification scheme [19] groups different types of

parallel processing systems depending upon their data and instruction organisations.

Single Instruction path Single Data path (SISD) represents a conventional computer design, where a single instruction path is followed, executing on a single data set. Most conventional microprocessors fall into this category. There is no real parallelism with these machines.

The Single Instruction path Multiple Data path (SIMD) class comprises many processing elements performing a single common instruction upon many data sets. Typical SIMD systems involve a large array of fairly simple processing devices, and these can perform low level image processing transformations at very high speeds [20, 6]. The image can be effectively mapped onto the array, so that each processor element works on the image section corresponding to its position in the array.

Multiple Instruction Single Data (MISD) systems represent pipeline processors, discussed above.

Multiple Instruction Multiple Data (MIMD) machines represent the most powerful and flexible parallel processing systems. Such systems, which include multi-transputer configurations, have the capability of working not only on different sections of a task, but also on different tasks, concurrently. Each processor in this type of network can communicate with other processors, allowing data and commands to be passed around the system.

Specific Parallel Processing Systems. The two more common parallel processing categories will be discussed here, SIMD and MIMD.

SIMD Systems. SIMD systems often comprise of a large array of simple processing elements. The array could consist of many hundreds of processing devices, each able to perform basic arithmetic and logical functions, store, load, and shift operations. Low level image processing transformations can thus be readily implemented, and often performed at very high speeds. The programming of such systems is greatly facilitated by the simplicity of the devices. This processor simplicity can be a disadvantage, however, because the programming of more complicated algorithms may be hindered by the need to use simple operations.

The load balancing of SIMD systems, when performing low level non data dependent image processing transformations is excellent, but would typically be far from ideal with more complicated algorithms. Different parts of an image require extremes in computation, some parts needing little, and others requiring a much greater amount of computation.

MIMD Systems. Several different types of MIMD systems have been proposed, mostly using conventional sequential microprocessors and shared memory, or shared buses. The performance of such systems and the number of processors that can be used, however, is limited by several factors. The bandwidth of the shared memory bus is not generally sufficiently high to enable more than a few processors to be used, and requires careful design to avoid undue timing problems [21]. Conventional parallel processing mechanisms must also be employed, involving Semaphores made up from Signal and Wait procedures, and Guards, to avoid shared memory usage violations and logical operation errors [22, 23]. The use of these overhead safeguarding mechanisms all serves to degrade the performance of shared

memory systems. Data and instruction broadcasting can also be inefficient due to these mechanisms.

Transputer Systems. A multi-transputer network does not suffer from having to use safeguarding overheads. Each transputer is autonomous, executes its own program, and has its own memory. There is no shared memory, and thus no shared variables between processors. There is no shared bus either, data and instruction broadcasting being achieved using high speed serial links, which can operate concurrently with the processor.

Transputers have essentially two modes of parallel execution. Many programs, or processes, can execute in pseudo-parallel on one transputer, with the processor time-slicing between them using a fast process switch implemented in silicon. The second parallel mode is where several transputers in a system execute in parallel, each executing its own program. Both types of parallel modes are usually employed.

Capabilities and Potential of Parallel Processing. Such is the potential of parallel processing systems that the performance of most applications can be increased dramatically. Many applications can be decomposed into a large number of sections, allowing the overall task to be distributed over hundreds or even thousands of processors.

A good example of such an application is that of Ray Tracing [24], which has been demonstrated by Inmos using hundreds of transputers. Each pixel of an image of an imaginary scene must be traced back to the scene's objects, and reflected many times, in order to evaluate its intensity and colour. Each processor must have a copy of

the scene's description. There is thus a small amount of data being passed around the system, and a large amount of non determinate computation associated with each pixel. The performance increase of these applications has been shown to be nearly linear with increasing numbers of processors.

Implementing Algorithms in Parallel. To implement an algorithm on a parallel system, such as a multi-transputer network, it is usually more effective to completely restructure the algorithm, often from first principles, than to try to extract parallel sections from it in its existing state. This procedure often leads to a far greater degree of parallelism being identified than could be found otherwise.

1.3 The Transputer Family

The transputer family consists of a number of compatible devices, including 16- and 32-bit processors, disk drive controllers, link adapters and switching devices [25], as shown in table 1.1.

The T414 and the T212 were launched in late 1985, and the T800 launched in late 1986. The enhanced transputer devices (T425, T222, and T801) were launched in early 1988. The next transputer to be launched is expected in 1990. The most commonly used transputer devices are the T414 and the T800.

Device	Description
T212	16-bit transputer.
T222	T212 with extra on chip SRAM.
T414	32-bit transputer.
T425	T800 without FPU.
T800	T414 with a 64-bit on-chip floating point unit.
T801	T800 with non-multiplexed address/data bus.
M212	T212 with Disk Drive Controller.
C001	Link Adapter, transputer link to 8-bit data.
C002	Link Adapter, transputer link to 8-bit data.
C003	Link Adapter, transputer link to 8-bit data.
C004	32 to 32 link crossbar switch device.
C011	Link adapter, transputer link to 8-bit data.
C012	Link adapter, transputer link to 8-bit data.

Table 1.1. Members of the Transputer Family.

All the transputers have a powerful processor, with an architecture similar to a Reduced Instruction Set Computer (RISC). The addition of fast on-chip memory, high speed serial communication links, a fast silicon process multiplexor, and other features make transputers very attractive processing devices (figure 1.7).

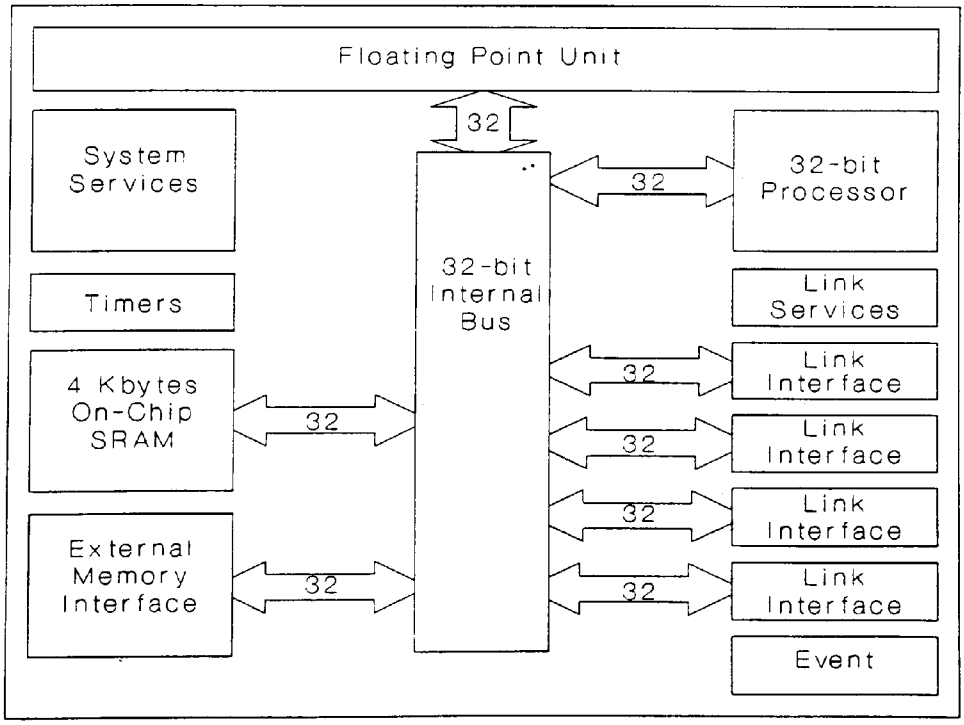


Figure 1.7. Block Diagram of the T800 Transputer.

The processors can be set to initialise or boot from ROM or from a link. Typically, in a development system, all devices would initialise from the host machine, via their links, but in a target application system, one processor would have an EPROM non-volatile memory holding the program for all the processors. This device would then boot from EPROM, and initialise the other devices with their programs using their links.

The T414 Transputer. This research project obtained some of the first T414s available, as this was the first transputer type introduced, and remains the cheapest. Some T800 devices were obtained later, although as most of the computation being performed within the project was not floating point intensive, they did not significantly benefit the research. Both the T414 and the T800 devices are full 32-bit processors, yielding 10 Million Instructions Per Second (MIPS) processing power at 20 Mhz on chip clock frequency. They have been benchmarked to be significantly faster than other 16- and 32-bit microprocessors [26, 27]. There are 4 full duplex high speed communication links, operating at 10 or 20 Million Bits Per Second (MBPS), allowing many transputers to be connected together to form large processing configurations.

The T414 has 2 Kbyte of fast access on-chip Static RAM (SRAM), and a configurable 32-bit wide External Memory Interface (EMI), allowing access to a linear 4 Gbytes of external RAM. Two on-chip timers allow time sensitive operations to be performed. Interrupts are catered for by Event input and output pins.

A very fast process switch, implemented in silicon, together with the small register set, allows the processor

to switch between processes in less than 1 microsecond (μS). This allows many processes to be time-sliced on a single transputer, simulating concurrency.

There are 3 system service signals, Reset, Analyse, and Error, which connect from one transputer board to another, in a daisy chain manner. The Error output allows program violations and other erroneous conditions to be passed to a host system, whereupon the occam source code responsible can be identified by a host system using the Analyse function.

The T800 Transputer. The T800 is essentially an improved T414, with a full ANSI-IEEE 754-1985 specification 64-bit on-chip Floating Point Unit (FPU), yielding 1-2 Million Floating Point Operations Per Second (MFLOPS) [28]. The T800 is actually faster at multiplying floating point numbers than it is at multiplying integers. This is because the integer mathematics are implemented in software, whereas the floating point operations are performed by the FPU hardware.

The fast on-chip static RAM has been increased to 4 Kbytes, and the serial link protocol has been improved, overlapping the data and acknowledge packets, resulting in significantly faster communication data rates. Special 2-dimensional graphics routines have been implemented within the microcode, for efficient block moves, copying and clipping.

The Serial Communications Link Protocol. The link protocol was improved on the T800 and T425 from the original T414 implementation. Figure 1.8 illustrates the original and the improved protocols. As can be seen, the improved

protocol allows the Acknowledge data packet to overlap with the Data packet, resulting in a significant speed improvement.

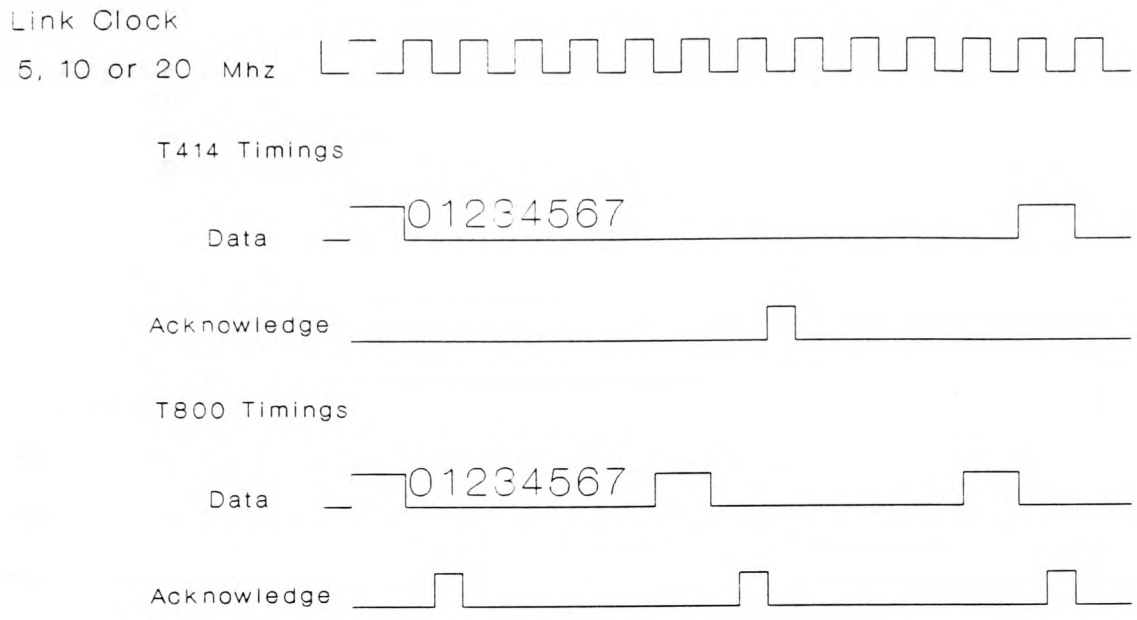


Figure 1.8. Serial Communication Link Protocols.

Future Transputers. As discussed in section 1.2, technological advances will allow the next transputer devices to have more facilities on-chip [12]. Referring to table 1.2, the percentage of the silicon chip area used by each of the major functional units can be seen for the T800 transputer. The remaining area, some 39 % is used up with miscellaneous items, such as the Event feature, interconnect and bus lines, and mounting contacts.

Functional Block	Area / mm^2	Percentage of Total Area
T800 Processor	93.0	100 %
Central Processor	12.0	13 %
Communications Link	2.5	11 % (for four)
Floating Point Unit	17.5	19 %
4 Kbytes SRAM	12.0	13 %
Arithmetic Logic Unit	0.6	0.6 %
70 Kbits Microcode	4.5	5 %
Remainder	36.4	39 %

Table 1.2. Area Usage of Major Functional Blocks.

If advances in technology allow the device to be manufactured in one eighth of the present area of silicon, say, and if the size of the silicon die is reduced by a factor of two, (to increase the manufacturing yield) then there will be effectively four times the area for extra new features. If the area used for the interconnect in table 1.2 is only reduced by a factor of two, to allow for increased bus loading problems, the effect of this extra silicon area can be estimated. Table 1.3 shows the projected area usage of the present major functional blocks.

Functional Block	Area / mm^2	Percentage of Total Area
T800 Processor	46.5	100 %
Central Processor	1.5	3 %
Comms Link (each)	0.3	2.5% (for four)
Floating Point Unit	2.2	4.7%
4 Kbytes SRAM	1.5	3 %
Arithmetic Logic Unit	0.1	0.2%
70 Kbits Microcode	0.6	1.2%
Remainder (Connect)	18.2	39 %
Available for New Features	21.2	46%

Table 1.3. Projected Area Usage.

Table 1.4 illustrates the possible new features that could be incorporated into a next generation transputer. Clearly the exact silicon usage will depend upon several factors, including market forces, future applications, and cost and ease of fabrication.

Feature	Increase in Area / mm^2
Increased on-chip SRAM from 4 Kbytes to 32 Kbytes	10.5
Extra 4 Links	1.2
Higher Speed Links	~ 0.7
Silicon Message Through Routing	~ 0.3
Increased Microcode	0.3
Debug and single step facilities	~ 1
Enhanced CPU	0.5
Enhanced FPU	1
Special Graphics Hardware (2D, 3D)	2
Improved Memory Management	1
Other Features in Silicon	2
Unused	0.7
Total	21.2

Table 1.4. Possible New Features on a Transputer.

1.4 Occam

Overview of occam. The primary programming language of the transputer is occam. The underlying principle comes from a 14th Century Philosopher, William of Occam (1280 - 1349), who's dictum was *Pluralitas non est ponenda sine necessitate* which translates to "One must not multiply entities without necessity", i.e. keep things as simple as possible.

Occam is a high level structured and procedural parallel processing language [29]. Occam compilers do support in-line transputer assembler coding, this facility being supplied to allow certain occasional specialised low level functions to be used from within occam programs [30]. It would not be envisaged that the assembler language would be used for any serious and substantial parallel processing application. The added complexity that this approach would involve would not assure the program integrity obtainable using fully checking compilers, and would not offer any substantial speed advantage.

Occam is a structured language in that all instructions are grouped together within constructs, delimited by indentation. Increasing the indentation of an instruction initiates a new construct, while decreasing indentation terminates a construct. Instructions at this indentation are thus within the former construct (figure 1.9). All variables have a definite scope of usage, which is the construct immediately following their declaration. Outside this construct their use is invalid, and not allowed by occam compilers.

```
SEQ  -- start of construct 1
  ...   initial instructions in Construct 1
  SEQ  -- start of Construct 2
    ...   instructions in Construct 2
    -- end of Construct 2
  ...   more instructions in Construct 1
```

Figure 1.9. Construct Delimitation by Indentation.

Evolution of occam. The first occam implementations were limited evaluation tools [31], allowing a user to gain some experience with the language, but this version, Proto-occam, was a small subset of the occam-2 language available at the time of writing. Occam-1 was the next language development, available on the M68000 based Stride microcomputer and the IBM PC and Compatibles, as an integrated Transputer Development System (TDS). The TDS included a novel and powerful Folding Editor, which facilitated the development of lengthy source code, and encouraged a structural "top down" approach.

The language was then improved, to occam-2, whereupon the IBM PC version actually executed on an IMS B004 transputer board plugged inside the host computer. Occam-2 was the full version of the language, featuring the various facilities discussed below. Finally, the product release of the TDS was made available, which included the full occam implementation, as well as many other features helpful to program development, including a Symbolic Debugger, Transputer Network Mapper and Tester, and enhanced library utilities.

Occam Primitives. Occam is comprised of primitive operators,for assignment, outputting an expression, and inputting into a variable (figure 1.10).

```
variable := expression        -- Assignment
channel ! expression          -- Output
channel ? variable            -- Input
```

Figure 1.10. Occam Primitives.

Constructs. Each occam construct has to have a construct execution identifier, that controls how that section of instructions are performed. The construct terminates when all the instructions or processes within it have terminated.

There is the conventional SEQ construct, for conventional sequential instruction execution, and the new ALT and PAR constructs, as well as a conditional IF construct (figure 1.11).

The ALT construct allows one of the subsequent constructs to execute, depending upon variable and channel conditions. This is most useful to selectively input from one of a number of possible channels, performing a channel multiplexor function. The selection of one process to execute is arbitrary if more than one process is ready to proceed, which can result in a program operating differently from one run to the next.

The PAR construct allows processes, at the appropriate indentation level, to be executed in parallel. All the processes within a PAR construct start executing simultaneously, and proceed concurrently. There is

practically no limit to the number of processes that may be defined, and executed concurrently. PAR constructs may have one of two priority levels, high or low, to allow short urgent or critical tasks to take precedence over other non-urgent more lengthy ones.

```
SEQ
  ...   perform task 1
  ...   perform task 2
  ...   perform task 3
```

(a) SEQ Construct.

```
IF
   condition 1
     ...   perform task 1
   condition 2
     ...   perform task 2
   TRUE
     ...   perform task 3
```

(b) Conditional Construct.

```
ALT
  chan1 ? variable
    ...   do task 1
  chan2 ? variable
    ...   do task 2
  TRUE & SKIP
    ...   do task 3
```

(c) Alternative Construct.

```
PAR
  ...   perform task 1
  ...   perform task 2
  ...   perform task 3
```

(d) Parallel Construct.

Figure 1.11. Formation of occam Constructs.

Construct Replication. All constructs may be replicated (Figure 1.12). Replicated SEQ implements the standard FOR...DO loops. Replicated IF enables elegant, correct and efficient searches to be constructed within arrays of data.

This method assures the integrity of the array search, eliminating boundary violations, and unnecessary searching. Replicated ALT allows channel multiplexing, for inputting from one of a number of possible channels within a channel array. PAR replication enables the execution of arrays of parallel processes to be specified.

```
SEQ i = 0 FOR 100
  ...   do task i

ALT
  ALT i = 0 FOR 100
    chan [ i ] ? variable
      ...   do task i
  TRUE & SKIP
    ...   do other task
```

```
PAR i = 0 FOR 100
  ...   do task i

IF
  IF i = 0 FOR 100
    condition
      ...   do task i
  TRUE
    ...   do other task
```

Figure 1.12. Replication of Constructs.

Variables. All variables are typed, with INT, INT16, INT32, INT64, REAL32, REAL64 allowing integer and real floating point arithmetic to be implemented. There are also BYTE, BOOL and TIMER variable types. A useful retyping function is provided that allows a variable of one type to be accessed as if it was a different type, in data manipulations and expressions.

Occam offers the use of multi-dimensional arrays, and can handle these in a very elegant and economic fashion. The whole array, or part of it, can be considered to be a single entity, for assignments, copying, and communications. Another feature, abbreviations, can also be useful, to effectively handle parts of an array as a

separate different array, for more efficient access, or use as actual parameters in procedures. Abbreviations can also be useful to make variables local to a construct, in which case the compiler can generate instructions to access the variables in a more efficient manner than would otherwise have been performed.

Channels. Parallel processes may not access shared memory space, and thus cannot share variables, for data transfers. This non-interference condition is essential to ensure correct parallel process operation. All communications between processes must use channels. Channels are unidirectional, and allow the transfer of data from one process to another. When two processes are to exchange some data, the first one that becomes ready waits for the other, then when they are both ready, the data exchange occurs. One process outputs data, in the form of a variable or expression, to the channel, and the other process inputs the data from the channel into a variable.

When concurrent processes are executing on one transputer, channels between them are implemented in RAM, but when the processes are on physically different processors, the channels are implemented in hardware, using the high speed serial links.

Placing Processes onto Transputers. Figure 1.13 illustrates the placing of three software processes onto one, two and three transputers using the PLACED PAR construct. The coding for each of the processes is the same in each case. The implementation of the channels between them, whether in RAM or on communication links, is transparent to the software.

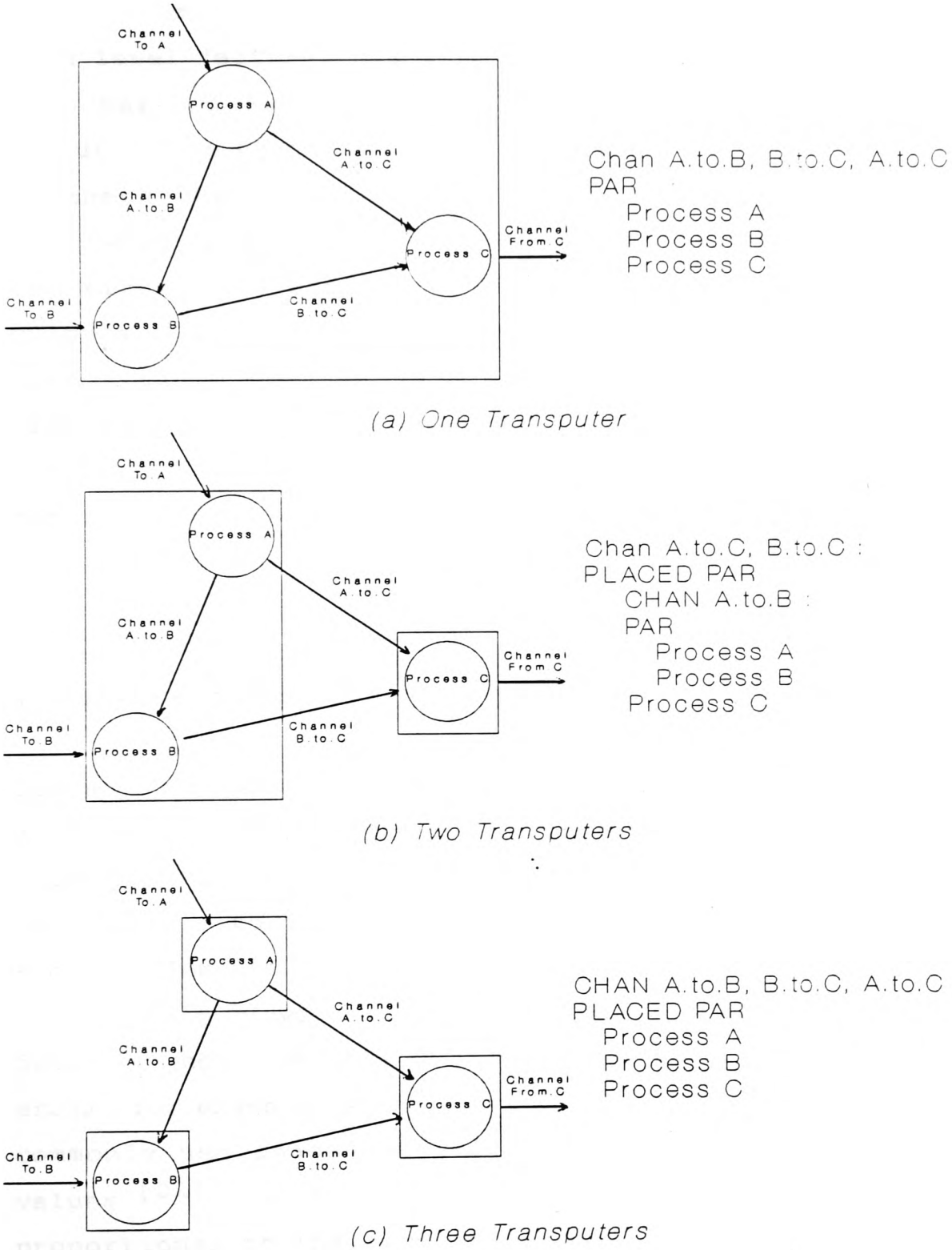


Figure 1.13. Placing Software Processes onto Transputers.

Scientific Functions and Other Features. Scientific libraries are provided in occam, allowing common trigonometric and higher level functions to be used. Other high level features are also present, including CASE OF, User Defined Functions, and Channel Protocols. It is useful to use Channel Protocols to allow the occam compiler to check channel usage and communications. A simple example of this may be the communication of a variable from one process to another. If the transmitting process outputs a BYTE variable, and the receiving process is expecting to input a 32-bit INT type expression, then clearly a serious problem exists. The second process would deadlock, and even if more data were subsequently sent, the communications will have broken down, with invalid data being received.

1.5 Image Processing

Images. Images are pictures obtained from a camera, and digitised into discrete binary numbers, representing the light intensity of each part of the picture. The image is digitised into rows of numbers, each row comprising of PIcture ELements (Pixels). The resolution of an image may vary, common sizes being 128 by 128, 256 by 256 and 512 by 512. Higher resolution images are used in specialist areas, for example, satellite and astronomical applications commonly use images comprising of $2 * 10^9$ 14-bit data values [32]. As the resolution of an image is inversely proportional to the size of a pixel, the smaller a pixel is, the more accurate a representation an image will be of a picture. The greater the resolution of an image, however, the more data there is to be handled and manipulated.

The intensity of parts of a picture, when converted into a digitised form, is represented by a binary number. Black is represented by zero, white by the highest value used, and the various levels of grey by the values in between. The range of the binary number yields the number of different intensity levels that may be represented. Common systems use 6 or 8-bit data to represent intensities, allowing 64 or 256 different light intensities to be used respectively. Specialist applications sometimes use greater precision data values, using 10, or 12 bit data, allowing 1024 or 4096 intensity levels, respectively, for a more accurate digital representation. As with the image resolution, the greater the precision of the pixel data used, the more accurate a digital representation will be, but there would be a corresponding increase in the volume of data required.

Image Data Sizes. If an image is of size *N* by *M* pixels, and the pixels have *B* bit data values, then clearly N * M * B bits have to be stored. An increase in any of these values results in a proportional increase in the amount of data used.

The use of binary intensity images to represent pictures can often be a useful way to dramatically reduce the amount of data used. A binary image consists of only zero or one, representing black or white, and can be coded using only 1 bit. The amount of data needed for this type of image would therefore be N * M bits.

Other image coding systems use colour, multispectral, stereoscopic views, or moving images [18]. All of these use different amounts of data to store images.

Image Processing Operations. Image processing operations are generally categorised into three levels, Image Enhancement, Feature Extraction, and Scene Interpretation [18], as shown in figure 1.14. The complexities of the operations, and the volume of data required at each level varies greatly. Image Enhancement incorporates simple operations on pixel values, repeated over the entire image. Feature Extraction involves identifying important features within the image, and extracting useful information about them. Scene Interpretation is when an intelligent decision process is made, regarding the contents of the image, using the extracted features.

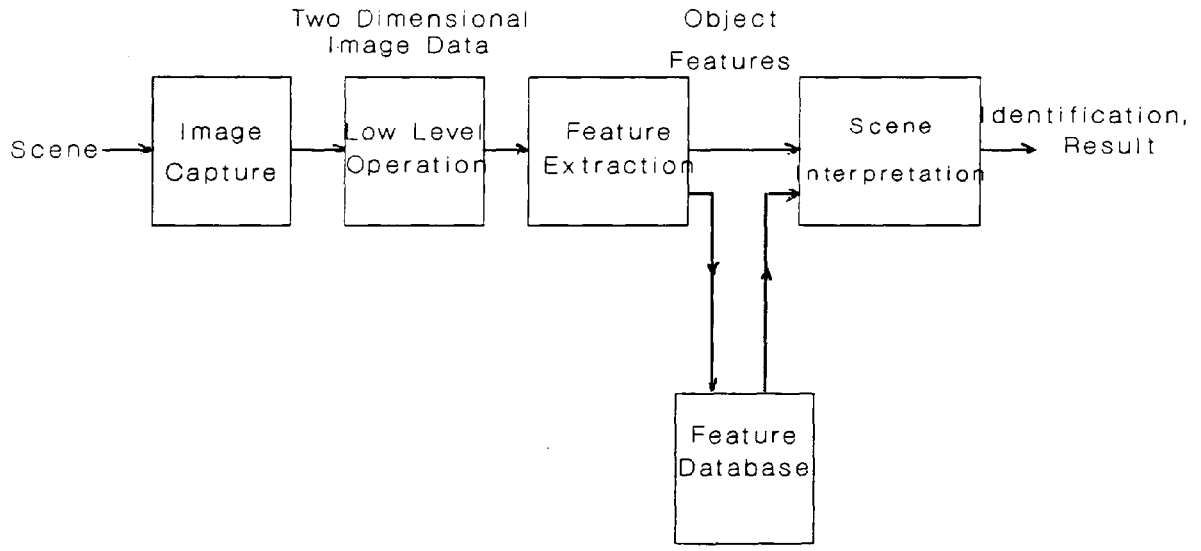


Figure 1.14. Image Processing Operations.

Image Enhancement. Image enhancement techniques are essentially low level, non data dependent transformations involving highly repetitive operations on large amounts of data. They are composed of Monadic, Dyadic operations and Local Operators, executing on one or two images.

Point processing or Monadic image processing algorithms at

this level typically involve the same operation being performed on each member element in the entire image. Monadic functions, on an image *A[]* of size *N* by *M*, may be represented by :

$$B_{[x,y]} := F (A_{[x,y]}) \qquad \text{where } x = 0,1,2...N \quad y = 0,1,2...M$$

Note: $A_{[x,y]}$ is the pixel value in image *A[]* on column *x* and row *y*.

Examples of common point processing Monadic operations are Intensity Shift, Negate, Bit Manipulations, Threshold, Double, Halve and Square Intensities, Image Multiply (by a constant), and Highlight Intensity Range.

A sub-class can be defined that require effectively two stages. The first stage is to determine parameters from the image, then the second phase applies those parameters to an operation over the image data. The second pass is identical in outline to a standard Monadic transformation, using the parameters extracted to subtly alter the transformation performed. Examples of this sub-class are Contrast Enhancement, Histogram Equalisation, and Automatic or Percentage Thresholding.

Many low level image processing operations, termed Local Operators, require access to the four or eight neighbouring intensity values of a pixel, when computing the new value for that pixel. Each member element in the image is replaced by some function of itself and the neighbouring elements within a window centred on that element. Common sizes of the neighbouring window are 3 by 3 and 5 by 5. For a 3 by 3 neighbouring window, and an image *A[]* of size *N* by *M* :

$$B_{[x,y]} := F\ (\ A_{[x+i,y+j]}\ ) \qquad \text{where } x = 0,1,2...N;\ y = 0,1,2...M;\ i,j = -1,\ 0,\ +1$$

Such operations include Dilation and Erosion, Gradients (directional and maximum), Edge Detection (Roberts, Sobel, Canny), Convolution and Filtering (High, Low pass, Median), Direction of Brightest Neighbour, Count White Neighbours, Largest or Smallest Neighbour, and Point Remove (White or Black). Local neighbourhood operations are also useful in smoothing images, to reduce the effects of noise, clean up, or blur the image. Faulty camera pixel values can be replaced by a local average.

Images can be geometrically transformed, including warping, shifting, and rotation. It is possible to compensate for non linear camera or lens defects, or transform the image to correct distorted views due to perspective angles. An example of this can be found in processing views of the Earth from orbiting satellites, which incorporate very wide angle lenses. The nature of this lens necessarily distorts the image, so some geometric compensation must be applied.

Dyadic functions involve the replication of an operation over the member elements of 2 images, *A* and *B*, of sizes *N* by *M* :

$$C_{[x,y]} := F\ (\ A_{[x,y]},\ B_{[x,y]}\ ) \qquad \text{where } x = 0,1,2...N;\ y = 0,1,2...M$$

Common dyadic operations include Add and Subtract Intensities, Minimum or Maximum Intensity, Exclusive-Or of the two images, and Boolean operations.

Feature Extraction. Feature extraction, by definition, involves extracting useful information about relevant

object features present within an image [33]. Features of interest include shape, size, moments, position, centroid, or intensity measures that characterise and describe objects or parts of objects in the image. In a typical shape analysis routine, an edge detected grey level image may be thresholded, so that only black and white pixels are present. Feature extraction algorithms must then explore the two dimensional image, looking for components such as edges and corners. Edge thinning and joining algorithms may be employed to improve the detected edges, after which the resulting improved edges may be inspected.

Edges may be chain coded, or objects run coded, to allow a more convenient and compact representation. The volume of data required to represent items of interest in the image is significantly reduced during this feature extraction procedure. Typically, with 128 by 128 or 256 by 256 images, the data may be reduced from 16 Kbytes or 64 Kbytes to a few hundred bytes.

Common feature extraction techniques employ Boundary Chain Coding, Run Coding, Convex Hull (CH) and Convex Hull Deficiencies (CHDs), Euler Number, height, width and position measurements and ratios, areas and perimeters, shape elongatedness factors, and moments.

Another feature extraction technique that can be employed is that of p-tuple processing, or template matching [9]. In this method a template is defined, based on some of the neighbouring pixel values, and used to generate information about the values of those pixels. The output of this template match procedure is a measure of how well the pixel and its neighbouring region match the predefined shape. This technique is commonly used in character recognition,

where binary lines and small areas are encountered and investigated.

Scene Interpretation. Scene Interpretation essentially involves some decision operation based upon the features that have been extracted from the scene under examination [34, 35]. This operation is entirely data dependent, and extremely computationally intensive.

Features extracted from an image may, in the simplest situation, indicate the presence or absence of an object. In this case the interpretation operation is a straightforward matter of deciding whether the object is present or not, and indicating the result appropriately.

Other features may be more complex, and involve distances, size measures, and computed size ratios or defect percentages. The Scene Interpretation required may involve determining an object's orientation, matching an object with previously encountered (learnt) objects, or deciding whether the object is acceptable according to some predetermined criteria. Other scene analysis tasks may be found in automatic vehicle guidance, object tracking, or speech and text analysis.

In the case of a matching process, the features of the object being inspected must be compared with the features of all previously encountered objects, so that the object may be classified correctly. Typically in real applications, however, an exact match may not be possible, in which case some strategy must be utilised to determine the identity of the object.

1.6 The Program Listings

For convenience and clarity, a number of occam simplifications have been made in the program segments shown. Two dimensional image section addressing is represented by A(x,y), rather than the correct occam method A[x][y]. Variable, constant and array declarations have been omitted, except where attention is to be drawn to a specific feature. Unnecessary detail has been hidden in *folds*, as illustrated in the Key to Abbreviations.

The number of rows and the number of columns in the image sections are represented by *no.of.rows* and *no.of.cols*, respectively.

1.7 Summary

The research project aimed to investigate software and hardware architectures for the implementation of image processing algorithms. Whilst low level image transformations can be performed at high speed by hardware logic, VLSI or DSP devices, these approaches cannot be used to perform the more complex feature extraction and scene interpretation operations. These require a programmable processor, to give the necessary flexibility, and parallel processing, to yield the performance increase essential to image processing applications.

The transputer is a new 32-bit computer-on-a-chip, manufactured by Inmos, at Bristol. It was designed from the outset, with the programming language occam, to effectively support parallel processing. On the chip there is a 32-bit Central Processing Unit (CPU), fast access

Static RAM, Four high speed communication links, and, with the T800, a 64-bit Floating Point Unit (FPU). Multi-transputer networks can be designed and implemented without the need for the complex and prohibitive shared memory or shared bus overheads traditionally required. The transputer is autonomous, each having its own exclusive memory, communicating only via its high speed serial links.

The use of occam enables highly parallel systems to be designed and specified, and correctly invoked and executed. Parallel processes may exist on a single transputer, and many such groups of processes may execute on many transputers. Occam, being a high level language, enables complex image processing algorithms to be realised, with a high degree of inherent software integrity.

Image processing is comprised of essentially three levels of complexity, Image Enhancement, Feature Extraction, and Scene Interpretation. Image Enhancement comprises repetitive, non data dependent transformations, including Contrast Enhancement, Convolutions, and Edge Detection. Feature Extraction involves data dependent operations involving the extraction of useful characteristics from an object of interest in an image. Scene Interpretation involves a decision making procedure, where the features already extracted are built into an object descriptor, and either matched to some predefined model, or used to determine the correctness of the object.

CHAPTER 2

SURVEY OF OTHER RELATED WORK

Many hardware and software configurations have been proposed in the literature for performing image processing operations [36,37]. Not all have been physically realised, however, and remain theoretical studies [38, 5]. A few research studies are underway [39], some supported by the ALVEY, ESPRIT and ACME programmes, to investigate the best architectures and algorithms for various image processing functions [40].

Many of the solutions that have been realised and published are discussed in more detail below, being sectionalised according to the implementation configuration, with due regard to the classification scheme widely used [19].

2.1 Other Surveys

A few survey papers have appeared, dealing with architectures for general multi-processor systems. The Intel iPSC, the discontinued FPS T-Series, Inmos Item 400, and the Meiko Computing Surface have been considered [41]. Processor pyramids have been discussed [42], where the top device is connected to 4 other identical devices, each of which are connected to 4 other devices, and so on. The PAPIA project was mentioned by Cantoni [43] which used 64 processors arranged in a 4 level pyramid. Burkhardt [44] investigated the execution speed-up obtainable with multi-

processor systems with different numbers of processors, and different problem types. Several problem areas were discussed which limit the speed-up obtained to less than linear.

Some survey papers have dealt with image processing systems and architectures. An earlier paper dealt with some architectures for pattern recognition and image processing [17]. Fu covered Unger's Module Interconnected Array from 1958, the ILLIAC III and IV, the CLIP series, PPM and PICAP, TOSPICS, the CDC flexible processor, and STARAN. These systems will be discussed later.

Batchelor [3] discusses 5 image processing systems suitable for real time execution of low level transformations. A Linear Array Processor (LAP), a pipeline based on the M68000 processor (KIWIVISION), a commercial hardware pipeline system (Maxvideo), the commercial implementation of AUTOVIEW on a PDP 11/23, and a M68000 are all compared for a variety of enhancement operations. It is concluded that arrays of processors must be used to gain the performance required to process low level image processing algorithms at acceptable speeds. It is pointed out, however, that arrays do *not* satisfy the requirements for higher level image processing. Transputer arrays are briefly mentioned, and the possibility of combining these concurrent processors with a SIMD array processor is indicated.

Elliott [7] compares arrays for image processing. Arrays of 8 by 8 transputers, the ICL MILDAP with 1024 processors, 4608 NCR GAPP devices, a DEC PDP 11/73, and a custom VLSI device are compared for executing a 3 by 3 convolution on a 512 by 512 image. It is stated that, for this low level

application, the large GAPP array achieves this in 10 mS, the custom device and the MILDAP achieve 20 mS, while the transputer array manages 30 mS. As a useful indication of comparative speed, the DEC PDP 11/73 performs the convolution in 20 seconds. Only a couple of low level image processing transformations were featured, however, so the implementation of more complex algorithms was not explored.

Krikelis considers the merits and constraints of VLSI implementations for low level image processing [45]. Advantages of cost, integration and density (and reliability) are weighed against disadvantages of regularity and (two dimensional) planarity. Regular arrays have problems loading and unloading data, as the number of edge data paths is typically $O(N)^{1/2}$ for an array of size N^2. The problems of shared memory using a common bus are briefly mentioned, and distributed memory systems are discussed. One and two dimensional array architectures are contrasted, in terms of image data loading and mapping, control, and silicon utilisation. The paper concentrates on low level image processing, and consequently makes no mention of the suitability or otherwise of regular structures for higher level operations.

2.2 Single Processor

Image processing has been implemented on several single micro- and mini-computers. A range of mainly low level image processing transformations has been implemented using compiled BASIC and assembler subroutines on an IBM PC [46]. Operation timings of the order of a few seconds were obtained. The somewhat lengthy times were due to the

processor used in the IBM PC, namely the i8088.
A Modula-2 development system executing on a DEC LSI-11/23 [47], enabled convenient program development and image processing algorithm realisation. The computer used was not really particularly suited to image processing, however, a fact reflected in the operation timings obtained, these being in the order of tens of seconds using 256 by 256 images.

The Illiac III computer was designed for scanning and analysis of specific types of data, and consisted of essentially three units [17, 48]. One of these units, termed the Pattern Articulation Unit, comprised of 1024 identical sub-units, called Stalactites, consisting of nine small stores, an OR-unit on the input, and an AND-unit on the output. Each stalactite could be connected to other stalactites in rectangular or hexagonal arrays, and could input data from and output data to each of these other connected units. Each stalactite in the Illiac III consisted of one circuit board, due to the technology level at the time, and with 1024 stalactites it was a very large machine, and was never actually fully completed.

The CDC Flexible Processor, a SISD processor, was designed for four channel radar applications [17]. Each Flexible Processor had a dual internal data bus linking the arithmetic unit, an array hardware multiplier, and specialised logic for square root and divide functions. This system could use several Flexible Processors together, communicating on a party line.

2.3 Dedicated Hardware Logic

High speed convolutions and look up table transformations can be performed using hardware and delay units, as shown in figure 2.1. For basic one to one transformation mappings, image data is presented to the look up table, and used to address the output data, which is then used in subsequent operations. Convolutions require that the input data stream is divided into three streams, one direct, one delayed by one image row length, and one delayed by two row lengths. In this manner, the logic unit effectively sweeps across the image, three rows at a time (once the initialisation phase has completed), producing convolved data. This technique would also be applicable to implementing Local Operators, using the neighbouring pixel values to address the output data.

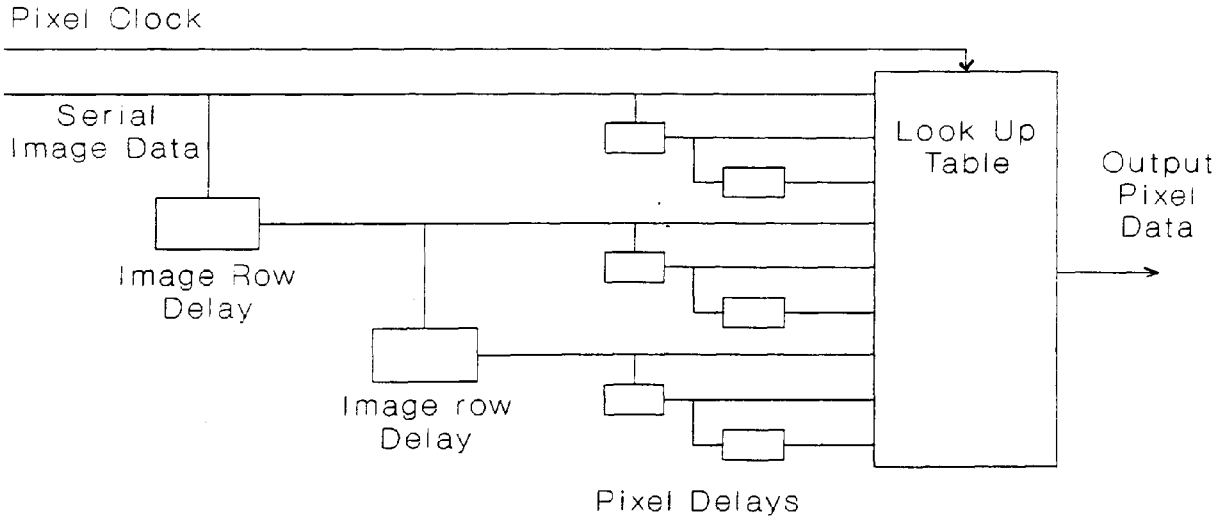


Figure 2.1. Convolutions and Local Operators Using Hardware.

One Alvey project [49] is investigating algorithms and design techniques for producing efficient processors in VLSI for various applications including real time image

processing. A specialised chip processor has been designed which will perform arithmetic and trigonometric functions in a few μS. An array of such devices is being considered, to enable the processing of television quality images. This should also be suitable for fast two dimensional convolutions.

There is a Wafer Scale Integration (WSI) project to investigate large two dimensional WSI processor arrays for performing tasks including image processing [50]. A second Alvey project aims to develop fault tolerant design methodologies for WSI [51]. Another Alvey project is investigating bit level systolic arrays [52] for tasks including signal processing & image coding.

A dedicated Real-time Imaging Module (RIM) was connected to an M6800 based EXORCISER, running M6800 assembler, with 6-bit pixels in 128 by 128 images [53]. Image processing transformation times of the order of seconds were quoted using the 8-bit processor. Although the paper does mention that the Imaging Module can perform transformations in real time, no results were quoted for this.

A Reconfigurable Attached Processor Architecture for Convolution (RAPAC) consisted of a dedicated hardware processor and multiple memory units, connected by a cross point switch under control of a host computer [54]. The architecture of the system could be altered slightly by the host, depending upon the task to be performed. A single pipeline configuration could be set up, or double pipelines used. Units included were frame stores, acquisition board, display board, and the processor unit, made up of a preprocessor for low level operations, and a feature extractor. Convolutions could be performed in 20 mS, using

256 by 256 images, Thresholding was performed using look up tables, and simple vehicle features could be highlighted in traffic scenes. Higher level processing could be performed by the host computer. This may be the slowest part of the system, however, and may not perform at the required speed.

2.4 Dedicated VLSI Devices

Local neighbourhood operations may be performed at high speeds by dedicated VLSI devices and quite simple circuitry [55, 57]. Such local operations involve the use of a weighting matrix (often of size 3 * 3) which is used in the summing of the local neighbouring pixels. Convolution, filtering, edge detection, erosion, dilation and point removal, for example, are all realisable using these techniques.

Arrays of VLSI devices for image processing are considered in [45], where the advantages of cost, integration, density are considered against the disadvantages of regularity, and planarity. Memory distribution and configuration architectures are discussed in this paper.

An array of VLSI processors with additional connectivity to allow broadcasting is considered in [1]. This allowed better broadcasting of data to the processor devices, using row and / or column connections.

The Map Orientated Machine (MOM) [58] is a dedicated system for image processing, combining the flexibility of von Neumann machines and the speed advantages offered by VLSI hardware. It consists of several units including pixel cache memory, data sequencer, and problem orientated logic

unit, connected via an interface to a host computer. The MOM aims to fall in between a fast (and expensive) SIMD systolic array, and a slow (but cheap) sequential von Neumann machine. The MOM can operate as a SIMD systolic array, but also has the ability to operate in a data dependent mode, moving across images in arbitrary directions to follow object contours etc.

2.5 Digital Signal Processing Devices

More complex low level image processing transformations can be implemented using Digital Signal Processing (DSP) devices. A common algorithm for image data compression, the Discrete Cosine Transform, has been considered for implementation on DSP devices in two papers.

The Multiply / Accumulate ADSP-1010 [59] and the IMS A100 [56] devices were considered in [2]. Another approach applied the IMS A100 as a co-processor to a transputer [60]. This implementation proposed an array of transputer IMS A100 processor pairs, using the transputer for control and data co-ordination purposes, and the A100 for the data intensive and repetitive signal processing.

The recently introduced DSP device, the IMS A110, is a cascadable signal processor, capable of performing two dimensional Convolutions, with programmable weights, at very high data rates [56].

Two systems featured in the literature have used the TMS32010 DSP chip. Holburn used a Z80A CP/M based host system to control a TMS32010 processor board [61]. The host processor could also access image data held in a frame

store. This allowed comparative timings to be obtained for the host Z80A and the target system. Low level image processing transformations involving Negate, Threshold, and Convolution were implemented, using images of 256 by 256, with the target system execution times being in the order of 300 - 1300 mS. The Z80A achieved times from 8.5 S to 40 S. The TMS32010 target system achieved moderate timings for the low level operations, but would not be suited to executing higher level, more complex image processing algorithms. The Z80A yielded an interesting timing comparison.

Ngan used eight TMS32010 DSP devices arranged in a SIMD configuration to implement some enhancement transformations on 256 by 256 images [62]. The host system was a MC68000 based Single Board Computer (SBC), and processing could either be performed by this host processor, or by the DSP devices. Sobel edge detection and line thinning algorithms were achieved with timings of the order of 106 mS and 580 mS respectively, excluding image data loading. The DSP devices all executed synchronously, which led to a simple system, but would make the implementation of complex image processing algorithms extremely difficult, due to the non-determinate data dependent decision process involved.

2.6 Pipelines

A Pipeline has been proposed using VLSI devices [63]. Four custom VLSI Image Pipelined Processor (ImPP) devices, the uPD7281, and a custom VLSI interface chip, the uPD0305, were developed, and used connected in a ring shape pipeline configuration. A range of image processing algorithms were implemented, with timings from approximately 40 mS to

800 mS being obtained. Character recognition was also achieved, using a predetermined element classification matrix and Euclidean distances from the feature vectors. This system recognised characters at approximately 50 characters per second (CPS), with a 99.3 % accuracy. The paper does not make the method clear by which this speed was achieved. It would appear, though, that this system would not be suitable for complex data dependent algorithms.

Modules made up of two 3 by 3 convolvers, look up table and local neighbourhood operation logic have been successfully used in a system named PIFEX [64]. Bit rates up to 8,000,000 12-bit pixels per second could be used. This design allowed the length of the pipeline to be altered depending upon the algorithm being executed. A system involving up to 80 modules was planned. The performance of the pipeline depended upon the task being executed, and while it was considered good for many transformations, other tasks may not be realisable at all. The machine was designed to efficiently execute low level image processing operations, and as such it would be difficult to implement more complex algorithms.

The Kiwivision pipeline processor [65] uses modules comprising of Arithmetic Logic Unit (ALU), look up table or local neighbourhood function units, allowing the overall structure to be varied to suit the application. Typical look up table transformations are performed on images of 256 by 256 pixels in approximately 11 mS [3], and local neighbourhood operations in 100-200 mS. It is apparent, however, more complex algorithms would take considerably longer to execute.

The Cytocomputer systems employ various pipelined configurations [66]. The later system described, Cyto-HSS, can perform sequences of several low level filtering and segmentation operations on a 512 by 480 image in 780 mS using 4 stages. The paper limited itself to the implementation of mainly low level image processing transformations, however, and it was not clear how difficult the execution of more complex algorithms would be.

Transputers lend themselves to straightforward pipeline implementations because of their inherent message passing ability and communications links. Transputer pipelines have been used in several computationally intensive applications, including Mandelbrot Set generation [67], Ray Tracing Graphics scenes [68, 24] and speech recognition [69]. The hardware configuration is simplified with a pipeline, as is the software architecture required in order to pass messages and data up and down the line of processors. Extra processors can be added to the configuration fairly easily.

Transputers have been proposed for the design of an adaptable pipeline [70], although the tasks that are discussed are general arithmetical problems, and not image processing operations. Elliott proposes a fault tolerant pipeline using transputers [7], although the paper makes no mention of this being used for image processing.

2.7 Closed Pipelines (Rings)

The Manchester Dataflow machine [71] architecture is essentially a ring of several processing modules, or

Function Units. Each of these modules performs a different task, operating independently in a pipelined fashion. The paper gives no direct timings for the image processing operations implemented, however, or any indication of the possibility of higher level algorithm implementation.

2.8 Node and Communication Network

Some systems proposed comprise of a number of processing elements, based on transputers, each of which can be connected to one of the others by means of a IMS C004 cross bar switch [72, 73]. This arrangement allows any processor to communicate with any other processor (after the appropriate link connections have been set up), but has not been used for image processing. The system would appear to suffer from the link switch set up overhead.

A multiprocessor system, named C.mmp, consisted of 16 processors sharing 16 memory units via a cross point switch [74]. Each processor could access any memory unit at any one time, providing the accesses were one to one, and not shared. Another system, named Cm, consisted of 50 DEC LSI-11 computers connected in an hierarchical structure [74]. Each processor had local memory that was also part of the shared memory in the system. These systems were not designed for image processing applications, however, and thus will not be extensively discussed here.

2.9 SIMD Arrays

The ICL DAP [75, 15] has 4096 simple processors each with 4 Kbits of memory, arranged in a regular array. The array processor is housed in a fairly large air cooled cabinet occupying several square yards of floor space. The host machine is one of the ICL 2900 mainframe computers, where the array looks like an ordinary memory module.

Often arrays are organised as Attached Array Processors [76] and used as an accelerating co-processing system attached as a peripheral to a host computer system.

The Parallel Processing Machine (PPM) and the modified form, PICAP were special co-processors for image processing [17, 77]. The PPM was essentially comprised of a neighbourhood matching logic, line buffers, neighbourhood counting register, and a co-ordinate register. The PICAP was dedicated to such tasks as fingerprint coding and parasite detection. The large amount of data in an image was reduced to an amount that could be more readily handled by a conventional computer.

The Toshiba company developed the TOSPICS interactive image processing system [17]. This used an image memory, with a parallel picture processor, to perform certain operations at high speed. Spatial filtering could be performed at the rate of 1 pixel/μS.

The Illiac IV, a large array processor was designed in the 1960s, having 256 processing elements divided into four quadrants. This was a form of MSIMD - multiple SIMD, as it could be structured as one or more SIMD machines [78, 79]. Each processing unit incorporated a sophisticated

arithmetic and logic unit, that performs floating point operations at high speed, allowing it to execute a wide range of operations. A certain amount of design mismatch is apparent, however, as many image processing transformations do not require extensive floating point mathematics, rather simple integer additions and subtractions.

STARAN built by Goodyear Aerospace in 1972 [17] was not specifically designed to be an image processing machine, but it was used in conjunction with a CDC 6400 host computer [74] for image resampling. An important feature of this machine was an associative memory array, providing content addressable abilities. The STARAN array consisted of 256 processing elements connected to a 256 by 256 bit multi-dimensional access memory. A conventional sequential host computer controlled the working of the array.

Unger's Module Interconnected Array consisted of an array of simple 1-bit processing elements [80]. A master controller broadcasted instructions, and each processor communicated with each of its four or eight immediate neighbours. Basic operations could be performed, including logical operations OR, AND, and NOT, shifting, and link / expand operations and tests. Although this arrangement could accomplish some quite complex image processing algorithms, the programming was apparently complicated and involved. The Module Interconnected Array architecture influenced the work of many later researchers, including Kruse's PPM [17], Slotnick's Solomon Computer [81], Duff's CLIP arrays [82, 20, 5], and the Illiac III [17, 48].

In 1967 a fixed logic 400 processor system, UCPR1, was demonstrated by Duff [83]. The array was configured as a

20 by 20 square, and it was used to find vertices of tracks in an charged particle input image. This led to a diode array being constructed [84], which further developed ideas about arrays of processors.

The CLIP Systems. The CLIP systems designed and built at University College, London, feature large arrays of simple processing elements for implementing various image processing algorithms [82]. In particular, CLIP3 [17], CLIP4 and CLIP4S [20, 5] CLIP7 [85, 5, 86] arrays have been featured in the literature.

CLIP (or CLIP1 as it is now called) was developed in the early 1970's [87, 5], following on from the UCPR1 [83] and the diode array projects [5], and this led to CLIP2 and CLIP3. CLIP1 consisted of a 100 processors in a 10 by 10 four connected square network. For this experimental array, one of three functions could be chosen to be performed by the network. CLIP2 consisted of a 16 by 12 array of hexagonally connected processing elements, each of which comprised two programmable boolean processors. These processors could independently execute one of 16 boolean functions between two input data bits. The two inputs were either from two binary images, or from one binary image and an OR'd connection of neighbouring signals. One of the two outputs from the processors was transmitted to the six neighbours, and the other was put into an image memory. By its very nature of operation, CLIP2 was limited to low level functions that did not require or generate any directional content, such as enclosure, expansion, and connection.

CLIP3 included the facility for dealing with directionality [17, 5]. The 192 cells could be connected

in a 16 by 12 array using a square or hexagonal configuration, and could now deal with grey level images. Each cell could accept input data from its neighbouring devices, and process these using a summing and programmable threshold unit, OR'd with a second image input, then two independent boolean functions could be performed by a boolean processor. There was also 16 bits of local addressable store. CLIP3 could perform image processing operations on large images by using a scanning technique, but this was some 3000 times slower than operating in the normal mode. Both the Illiac III and the CLIP3 array suffered from edge effects caused by moving window operations over the image [17].

The CLIP4 utilised Large Scale Integration (LSI) technology to put eight slightly modified CLIP3 cells on a single N-MOS silicon chip in a four by two block [17, 20, 5]. The CLIP4 array has 16 by 12 identical cells, which can be connected as with the CLIP3 cells. The boolean processor can implement two of 16 boolean operations on the two inputs. There are basically three types of operations, Load, Process, and Branch. Instructions that do not involve any propagation through the array execute in 1 μS. Those that involve propagation depend upon the cell array size and the size of the image. A commercial version of the CLIP4 system was produced.

The use of the CLIP4 system has suggested two areas that were addressed with the design of a CLIP7 system [85]. While it is now possible to form many processing devices on a single silicon chip, advances in image resolution requirements continue to outweigh the number of processors offered using this technique. Scanning techniques were used with the CLIP3 array, to process larger images than

the 16 by 12 processor array. The CLIP7 design also attempted to address the problem of processing the image in a non-geometric way, in which processors could operate independently of each other. The CLIP7 system had a 512 by 4 processor array, which could be vertically scanned across a 512 by 512 image. Each processor had to perform the operation 128 times, on the different parts of the image. CLIP7 achieved execution times in the order of a few mS for simple arithmetic, boolean and local operator transformations.

All the CLIP systems, however, while efficiently executing low level, highly repetitive transformations at high speed, would be less effective with more complex algorithms. The programming of such algorithms would also be complicated and lengthy.

The Partitionable SIMD/MIMD machine (PASM) [88] could be partitioned to function as many independent SIMD and / or MIMD machines. Specifically designed for experimenting with image processing and pattern recognition tasks. It was attached to a host computer (PDP-11), for system control. The structure consisted of 1024 M68000 processor-memory units. Some considerations for the implementation of higher level image processing algorithms on this machine are in [89].

The German Suprenum machine embodies SIMD & MIMD techniques [90]. A system may consist of 16 clusters using toroidally connected communications buses (hypercluster rings). Each cluster has 16 nodes on two shared buses, together with disk and file server, communications controller, and housekeeping functions. This machine was not designed for image processing, however.

The Geometric-Arithmetic Parallel Processor (GAPP) device [6], invented by W. Holsztynski, enables large SIMD arrays with tens of thousands of processors to be constructed, allowing low level image processing operations to be carried out at very high speeds. Typical times obtained were 6 mS for edge detection (with thresholding), skeletonisation and labelling, using a 256 by 256 image. These timings represent the fastest low level image processing transformations that the author has come across. The implementation of more complex algorithms, however, may be difficult to achieve.

A Massively Parallel Processor (MPP) was built by Goodyear Aerospace for NASA, being completed in 1983 [91, 74]. It consisted of 16384 fairly simple processors, and was designed for satellite image processing. The processors were connected in a 128 by 128 square arrangement, operating in SIMD mode, with each processing unit able to exchange neighbouring pixel values with adjacent units. Each processor could perform the basic arithmetic functions at high speed, and had 1 Kbyte of local memory. A minicomputer, a DEC PDP-11, performed the program and data management control during execution of the machine, and allowed software development. A DEC VAX-11/780 was used as a host computer, to allow user interaction and MPP control. The image processing operations implemented were mainly low level transformations associated with satellite images. The system was not designed to perform complex data dependent algorithms.

2.10 MIMD Arrays

Arrays of transputers have been used for image processing [8, 92]. An array of 16 transputers arranged in a 4 by 4 square has been used to implement several enhancement type image processing transformations. Simple contrast enhancement was achieved in 124 mS, and convolutions using a 3''by 3 mask in about 150 mS, both using a 256 by 256 image.

The Disputer system [16] used a SIMD processor array (the Disarray) in conjunction with a MIMD array comprised of transputers. The communication and control between the MIMD array and the SIMD processors was achieved by memory mapping the control registers of the SIMD processors into the memory space of one of the transputers, which then acted as the overall controller. A limited amount of image processing has been implemented upon the Disputer, however, as most of the tasks that have been realised have been associated with graphics.

Kodak have used a toroidal array of transputers to implement image processing [93]. Each of the processors at one edge of the array had access to some dual ported memory, to enable efficient image data distribution. The dual ported memory was accessed via a VME bus from a host computer, for loading and unloading images. A control transputer, also connected to the VME bus, allowed instructions and data to be communicated to the array, and for the initial program booting to be performed. Several image processing algorithms have been implemented on the system, with processing times from 60 mS to several seconds being achieved using 512 by 512 images, and 32 transputers. This system clearly performs low level transformations

effectively, due to its regular configuration, but the paper does not discuss the potential of the system for executing more complex data dependent image processing algorithms.

Meiko have designed a commercial system, The Computing Surface, comprised of a configurable network of transputers [94]. This has not been featured extensively in the literature, however, with an image processing application.

Transputers were used in an array configuration in the T-Rack, developed under the Alvey ParSiFal project, and used to perform some image processing tasks [95]. Low level transformations and simple object tracking were implemented. The timings produced for the low level operations were somewhat slower than might be expected, using an array of 64 T800 transputers. Edge Detection using the Roberts or Sobel operators were achieved in approximately 100 mS and 275 mS respectively. The simple object tracking used monitored edge points, and achieved real time performance.

2.11 Shared Bus Systems

Conventional processors have been used in shared bus tightly coupled multi-processor systems [96]. These have been demonstrated to be effective for small numbers of devices, but with larger systems the memory data bus bandwidth and necessary overhead protocols [22] degrade the overall performance [23, 21].

2.12 Shared Memory Systems

The Paracomputer (also called the WRAM or Ultracomputer) is a theoretically proposed MIMD machine consisting of many processing units, each able to access a large memory using a common interconnection network [74]. Each processor can also have a local memory. Some of the machine's theoretical operating characteristics may be difficult to realise physically, however. An example of this aspect is that all the processors are (theoretically) able to access any part of the global memory simultaneously. This clearly represents severe memory and bus contention, and would not be possible (if at all) without significant performance degradation due to traditional shared resource control [22, 21].

The IBM Research Parallel Processor Prototype (RP3) employed shared memory and message passing paradigms, determined statically and dynamically [97]. The machine architecture comprised of 512 32-bit proprietary design processors, each with 2-4 Mbytes of memory, operating in an MIMD mode, and arranged in a rectangular SW Banyan network. It is not clear from the paper what tasks will be implemented on this machine, however, so the implementation of complex image processing algorithms cannot be discussed at this stage.

2.13 Tree Configurations

Tree configurations are being used for image processing [98] and graphics [99] (figure 2.2). The Radon transform was implemented using a ternary tree architecture transputer network by Hall. This consisted of distributing

horizontal and vertical rows and columns of image data to worker processors in the tree, for the Radon transform to be applied. The tree architecture was adopted for the implementation of the Radon transform for some of the reasons proposed in Chapter 5.

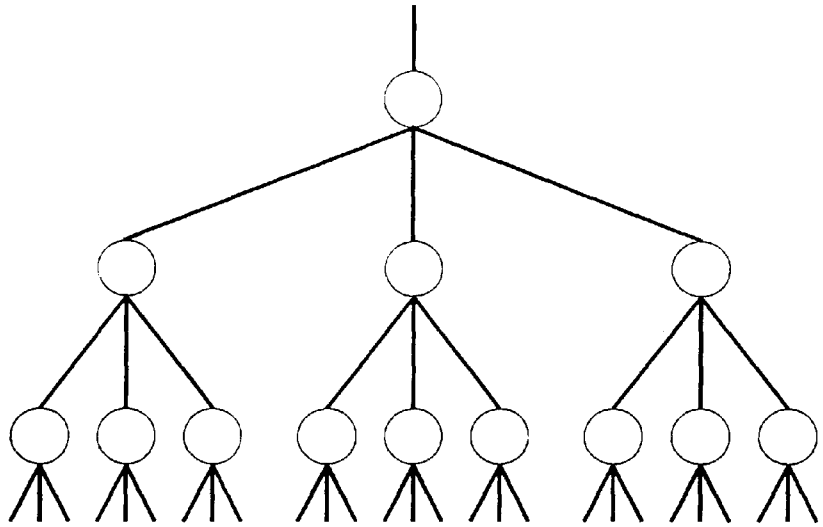


Figure 2.2. Ternary Tree Configuration.

2.14 Hierarchical Trees

The Hierarchical Array of Microprocessors (HAM) consists of an inverted pseudo-ternary tree of processing devices for image processing [100]. Image data is processed at the bottom of the tree, then features of objects are coded and dealt with at the other layers of the tree. A gated shared bus allows data from the pixel level processors to be passed up to the next level of devices.

Transputers are considered for use in an inverted ternary tree structure by Ham [101] although only theoretically. Several considerations are discussed when dealing with image data at television frame rates. As the proposals are only theoretical, however, no system had been implemented, and as such no timings are available.

NON-VON described by Ibrahim [102] is a SIMD inverted binary tree structured machine for image processing. Image data is processed at the lower level nodes, as described by Bourbakis, Miller [100, 103]. Image correlation, shifting, and template duplication have been implemented.

2.15 Pyramids

A pyramidal approach to image processing is proposed by Miller [103]. Here image pixel data is processed by the lowest level processors in the pyramid, as in Bourbakis, Ibrahim [100, 102]. This structure is effectively an inverted order four tree, with additional communications links between the processors at each level (figure 2.3). Each horizontal level is a square array. Operations involving nearest-neighbour algorithms are presented, as well as a Convex Hull algorithm. The pyramid structure is proposed as an efficient configuration for image processing due to the inherent hierarchy which is analogous to the different levels of complexity of operations used in an application.

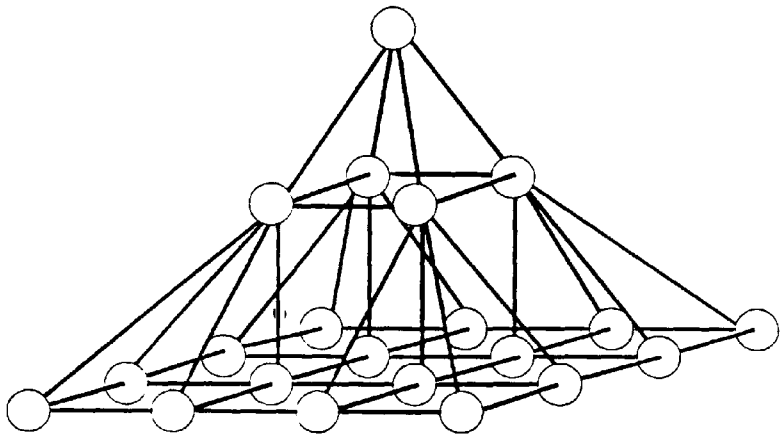


Figure 2.3. Three Level pyramid Configuration.

A different type of processor pyramid is considered by Sandon [104]. With this SIMD configuration, an N by N array of simple processor devices can be reconfigured into any one layer of the overall pyramid. While the number of processor nodes is decreased on each layer towards the top, the number of processors in each layer is the same. This means that for all layers excluding the bottom one, there will be successively more processors at each node. Clearly this arrangement does mean that only the layer that the processor array has been configured for can execute at any one time, although if more than one array were used, several layers could execute concurrently. If a simple type of cascadable processor device was used, such as a single bit processor, then the multi-processor nodes would become more powerful, being 4-bit, 16-bit or 64-bit processing nodes. This was considered useful, to offer a greater degree of computational power at the higher levels, which is where it would be required. The paper specified the single bit processor that would be necessary, in order to be able to concatenate them into longer data length processor nodes. The paper does not consider the implementation of any actual higher order image processing

algorithms, however, so the effectiveness of the structure and the ease of programming is not apparent.

A pyramid structure was also considered for image processing by Merigot [105]. Termed the SPHINX machine, it was an inverted three dimensional binary tree, operating in an MIMD mode. The paper did not describe the implementation of any image processing algorithms in detail, however, so it is difficult to determine the suitability of such a system for complex algorithms.

Pyramid Network using Transputers. Transputers have been considered for use in a pyramid architecture for knowledge based sonar image interpretation [106]. The pyramid configuration proposed will use 21 T800 transputers in three hierarchical layers. The base layer has 16 processors arranged as a 4 by 4 array, each connecting to one of the four processors (in a 2 by 2 array) in the middle layer. The top layer is a single transputer, connecting to each of the four devices in the middle layer. This arrangement requires a minimum of six communications links, and as transputers have only four (at the time of writing), the extra links may either be provided by memory mapped Link Adapters, or multi-transputer nodes, as discussed in section 1.2 above, and [107].

The base level of the pyramid may also incorporate some custom hardware, to enable each transputer in this level direct access to image data. Each layer in the pyramid corresponds to a different level of the image processing being carried out. The bottom layer will perform low level image enhancement and segmentation, the middle layer will extract features, and the top layer will perform the image interpretation task, dealing with an abstracted symbolic

object representation. There may be problems in the distribution of the workload, however, due to the fixed number of processors at each level. It may be found that a single device is not sufficient at the apex of the pyramid to perform the image interpretation task. It would appear that this structure is best suited for performing image analysis on repetitively captured images, so that while one layer performs its level of image processing, a higher layer could be performing its function on the previous image.

2.16 Reconfigurable Network

The Reconfigurable Chain-structured Butterfly ARchitecture (RECBAR) [108] allows thousands of processors to be connected with a low connection cost. The first network described uses four communication ports per processor, (which could use and benefit from a transputer implementation) and the second one uses six. Fourier Transforms are the only image processing operation discussed in this paper, however; as the main consideration is the hardware structure.

2.17 Butterfly Network

A Butterfly network using transputers is considered by Taylor [109] to implement Fourier Transforms. The advantage of this configuration is that the transformation directly maps onto the network, with processor connections directly implementing the appropriate data paths. It can be possible to achieve super linear performance speed-ups with such arrangements, as a significant proportion of the

data indexing and selection is inherently performed by the hardware communications connections. This dedicated configuration would, however, be unsuitable for general image processing algorithms due to the arrangement of the links.

2.18 Hypercubes

Several systems have been built using a Hypercube configuration (figure 1.4). These machines have not been used for image processing, however, but are discussed briefly here because they represent interesting and large arrangements of processors.

A seven dimensional hypercube, The (Caltech) Cosmic Cube, [110, 111] utilises 128 processing elements constructed modularly from five dimensional 32-processor blocks. This follows on from the previous six dimensional 64-node Cosmic Cube hypercube. Each processor node has a 8086 / 8087 pair, 256 Kbytes memory, 32 Kbytes EPROM, and 7 full duplex buffered asynchronous communication channels. An Intel 310 system acts as the host computer. The system software uses concurrent processes in each individually identified node.

The Intel iPSC/2 is a 2nd generation hypercube [112, 113] which can be configured with between 4 and 128 nodes, with a 80386 / 80387 processor pair and up to 16 Mbytes memory at each node. A Direct-Connect message routing system enables the hypercube to be programmed as if all nodes were connected to all others.

The Connection Machine [114] features 65536 simple

processors, each with 4 Kbytes RAM. The basic structure is that of a 12-dimensional hypercube so that each node is connected to the other 11 nodes. At each node there are 16 processors, connected to a switching router, so that each processor can communicate with one of the other processors at that node.

2.19 Commercial Image Processing Systems

There are numerous commercially available image processing systems, most of which use a conventional host computer, resulting in a cheap but slow system. Some use dedicated VLSI devices or SIMD arrays to achieve respectable transformation timings. Many of the systems are very general purpose, though, using hardware, and as such the algorithms implemented tend to be mainly low level.

AUTOVIEW is an interactive image processing facility executing on a DEC VAX or DEC LSI/11 MINC minicomputer [115, 116]. The system only used the host computer, with an impressive repertoire of image processing algorithms implemented entirely in software. Consequently image transformation times were slow, and complex algorithms executed even more slowly, being in the order of tens of seconds. The system was designed to be a development tool, however, so the relatively slow performance was not of critical importance.

Amplicon Electronics Ltd., market a high performance modular image processing system [117]. This consists of a number of modules including a pipeline processor (Barrel Shifter, look-up tables, four DSP devices), real time

convolver, histogram and feature extractor, and an image processing accelerator. Real time convolutions could be carried out on data, and some simple data value features extracted.

One commercial image processing system is implemented in software, executing on an IBM PC or compatible [118]. This system offers an extensive selection of algorithms, but as all execution is performed on the host computer, using the i8088 or i80x86 microprocessor, the timings achieved are slow.

Several systems incorporate dedicated DSP or VLSI processors, either singly or in small arrays. The MDP-4 system [119], marketed by CDA Ltd., uses a programmable four processor array, achieving 3 by 3 mask convolutions with a 512 by 512 image in 500 mS. The system interfaces to a DEC MicroVax minicomputer system.

A similar system, also by CDA, uses an M68020 and a floating point vector processor, to enable high speed computation of floating point intensive applications [120]. Complex Fast Fourier Transforms (FFTs) involving 1024 points can be executed in 4 mS, and a complex two dimensional FFT can be applied to a 512 by 512 image in 2.5 S.

Caplin Cybernetics market a transputer based system that can be used with a DEC VAX minicomputer, for performing various applications including image processing [121]. No details of the software or timings for the system were available, however, but it is thought that the system is effective for low level, non data dependent algorithms, but not for complex operations.

Datacube market a range of image processing boards termed the Maxvideo Modular system [122]. These boards offer different capabilities, to enable an image processing system to be built up for any given application. Modules are offered for transformations including real time Convolutions with 512 by 512 images, neighbourhood pixel operations, and Histogram manipulations. Simple Feature Extraction is also possible, and a library of routines is provided to cater for other algorithms. These latter algorithms execute on a general purpose array processor, and consequently do not achieve real time operation.

KGB Micros Ltd., market a software image processing system called Image-Pro, executing on an IBM-PC, PS/2 or compatible [123]. The package offers a selection of low level transformations including Convolution, Image Arithmetic, Histogramming, and Enhancement. The execution timings are determined by the host computer, however, and will therefore be extremely slow compared to other systems.

An image analysis system utilising the TMS 320 series DSP devices is available for the IBM-PC or compatibles [124]. This system uses one or two plug in boards to allow image operations to be performed by the DSP devices. Image processing functions including Binary operations, Convolution, Erosion and Dilation, and Adaptive Thresholding can be applied to 256 by 256 images. A Convolution using a 5 by 5 weighting mask can be achieved in approximately 20 mS using a third optional Convolution Board.

A Frame store and associated software for various low level image transformations is available for use with a BBC Microcomputer [125]. Images of 192 by 256 using 64 grey

levels can be used, although as all image processing is performed by the BBC host computer (using the M6502 microprocessor), image transformation timings are slow.

A M68000 modular system, Visionix, allows the industrial use of image processing [126], featuring Dimensional Inspection, Form Recognition, and Detection and location of defects. Processing is performed by the M68000 devices, connected together using a VME bus.

2.20 Summary

Many parallel processing systems have been proposed and used for a variety of image processing tasks. Some single processor systems have been designed, with algorithms implemented totally in software executing on the host processor. These tend to have extremely limited performance, however, due to the low computational power offered by the host processor. VLSI and DSP devices have achieved fast timings for low level transformations. Of special interest are the new type of cascadable DSP devices, which can perform two dimensional Convolutions at very high data rates.

Arrays of simple processors have been widely used. These have been successful in the execution of non data dependent, data intensive transformations. Execution timings have been obtained, using a very large SIMD array of simple GAPP processors, in the order of 6 mS, for Edge Detection (with Thresholding), Skeletonisation, and Labelling, using a 256 by 256 image [6]. These represent easily attainable real time execution.

Other configurations proposed have interesting potential, especially Hierarchical Pyramids, where the levels of image processing map onto the levels of the pyramid. A Pyramidal Structure has been proposed using transputers, which could, if extended, use different numbers of transputers at the nodes of each level, in order to match the processing requirement to processor resource.

Reconfigurable systems may, in theory, yield the best configuration for the level of processing being performed. Arrays could be used for transformations, and perhaps trees or pyramids used for the higher levels, but more work needs to be done in this area.

Highly parallel systems using simple processors, however, cannot easily perform data dependent image processing operations. A programmable component is essential, in order to achieve the flexibility necessary for these complex algorithms.

Commercial image processing systems mainly use DSP devices to perform the low level transformations, and the host processor, to perform any higher level algorithms provided. The author is not aware of any commercially marketed system that uses parallel processing to perform anything other than non data dependent algorithms. This is clearly due to the fact that much work remains to be done to exploit the true MIMD parallel processing technology for complex image processing algorithms.

CHAPTER 3

LABORATORY EQUIPMENT SETUP

3.1 Introduction

The transputer image processing laboratory essentially consisted of the video equipment, a Transputer Development System (TDS), and a target multi-transputer system, as shown in figure 3.1.

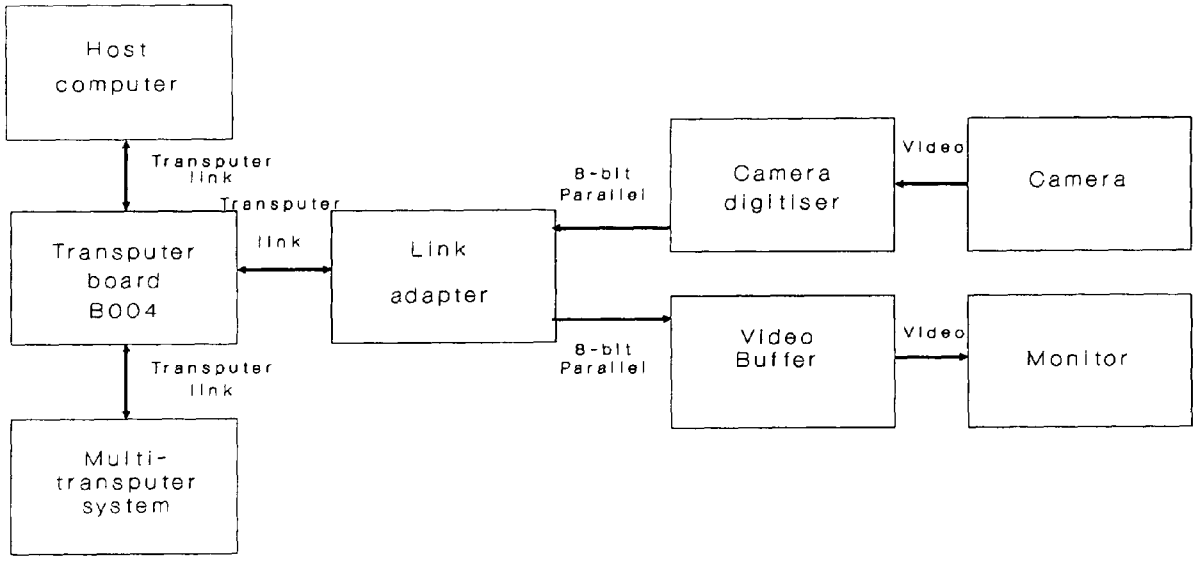


Figure 3.1. Laboratory Equipment Setup.

Interfaces were devised to allow transputers to capture images from a camera and digitiser pair, and to display images on a video monitor.

Software could be developed and compiled on the host microcomputer, then downloaded and executed on the target

transputer network. Images could be captured dynamically from the camera, transformed images displayed on the monitor, and numerical results and parameters displayed on the host computer's screen

3.2 Evolution of Equipment and Development Software

Initial Equipment. Some equipment was inherited from a previous project, including a DEC LSI/11 MINC minicomputer, a 128 by 128 camera and digitiser, and a video monitor and driver buffer. The funding of the research did not allow the camera and video equipment to be updated, as it was decided to give priority to the purchase of an IBM PC Compatible, IMS B004 transputer board, and transputer hardware.

The M68000 Stride Host. A M68000 based Stride 440 microcomputer was initially used for software development, using the occam-1 version of the Transputer Development System (TDS). Occam-1 code could be compiled for, and executed on either the M68000 of the Stride, or an external IMS B002. The TDS on the Stride was later upgraded to feature occam-2 and enhanced development utilities.

The PC Host. Inmos subsequently adopted the IBM PC Compatible as their primary host computer, due to market forces and the widespread use of this microcomputer. Software support for the Stride was phased out, with all new utilities and tools being developed for the PC version of the TDS. Consequently an appropriate microcomputer was purchased (an OPUS III), and the required internally fitting IMS B004 transputer board.

The Transputer Development System Software. The TDS now executed on the IMS B004, with a server program running on the host to allow the TDS access to the host facilities, including the screen, keyboard, and hard disc (figure 3.2). The actual product version of the Transputer Development System for the PC was released in 1988, replacing the beta version used previously. An enhanced set of utilities and tools were then available, including a transputer network mapper and tester, and a symbolic debugger. The use of the debugger will be discussed in more detail later, in the software development section.

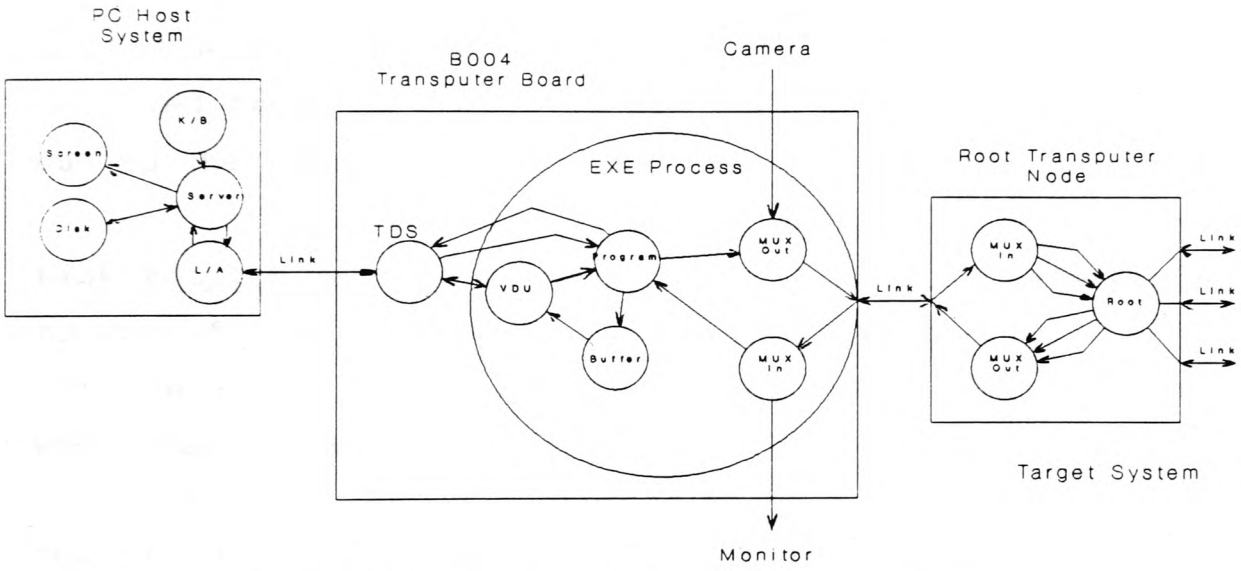


Figure 3.2. Host Facilities Access from TDS and Target System.

3.3 Camera Digitiser and Monitor Interface

Z80 Interface. An initial interface using a modified Z80 based Single Board Computer (SBC) allowed images to be displayed on the video monitor from the Stride microcomputer [155]. Two RS232C serial communication ports on the Stride were used in parallel, running at 38,400 baud, to transmit data to the Z80 SBC. A second RS232C serial port was mapped into the address space of the

Z80, and physically attached to the board. The baud rate of the serial lines was increased by using a frequency derived from the main Z80 processor clock.

A Parallel Input and Output (PIO) device was used on the Z80 SBC to strobe this data into the video buffer driver unit. A small program was written in Z80 assembler, residing in EPROM on the Z80 board, to initialise the two serial lines, receive the data, and output it to the display. The 128 by 128 image used 16 Kbytes of data, this being transferred to the Z80 SBC in 2.3 S. This allowed image processing transformations to be investigated before a link adapter became available. Images could not be received from the camera using this arrangement, however, due to the insufficient speed of the Z80 processor.

Link Adapter Interface. A link adapter IMS C003 was used to interface a transputer to the camera and display [107] (Figure 3.1). The link adapter and associated circuitry was housed on a double sized Eurocard.

The circuitry required to transmit images to the display using the link adapter was straightforward. The 8-bit data from the link adapter was buffered using a simple TTL 74 series device, before connecting to the video buffer driver. The output signal *QValid* was used to strobe this data into the video driver.

The camera digitiser used was asynchronous, free running, and had no handshake controls. Data was output from the digitiser, and strobed into the link adapter using a data valid signal. Extra circuitry had to be designed to ensure that invalid pixel data was not strobed into the link adapter at the ends of displayed lines of pixels, and also

during the camera end of frame beam fly back time.

In order that only complete images from start to end were received by a transputer, extra circuitry had to be designed that identified the start of an image. Due to the timing requirements of the digitiser, it was not possible to identify this period using software. Furthermore it was essential that a transputer could receive an image whenever one was required, and not to have to receive all images from the digitiser continually.

After initial power on or reset, the extra logic allowed one byte of image data to be strobed into the link adapter, whereupon the link adapter would then wait for the acknowledge packet from the transputer. The flip-flop would then be set, disabling any further data strobes from the digitiser. This state would continue until the transputer transmitted a byte to the link adapter. This would reset the flip-flop and the logic so that the next start of an image would cause data strobes to be gated through to the link adapter, starting with the first byte of the image.

In this manner, a transputer could input an image at any time, simply by outputting a single byte (to reset the flip-flop and logic) then initiating an image input sequence. The average delay before receiving the starting data of an image after outputting the single byte is approximately half the image time, determined by the digitiser and camera pair. The total image time was initially 90 mS, but once the system was working, it was possible to increase the internal clock of the digitiser, to reduce this value to 50 mS. The average frame synchronisation delay was thus 25 mS.

Use of a link adapter enabled a higher data rate to be achieved when displaying images than was possible with the Z80 interface. An image of size 128 by 128, i.e. 16 Kbytes could be transmitted in approximately 41 mS using a 10 MBPS link speed on a T414B. This could be reduced to approximately 15 mS by using the enhanced link communications protocol available on T425 and T800 devices, at 20 MBPS.

3.4 Quad Transputer Board

A Quad transputer board was designed and constructed for use with the research project. This is discussed in detail in Chapter 4.

3.5 Transputer Board Rack

A 19" rack was constructed to house the link adapter and quad transputer boards. This contained a 5 V power supply, to make the unit self contained.

3.6 Software Development

All the image processing software was written in occam-2. The Transputer Development System used allows software to be written, checked and compiled, then executed on the IMS B004 plug in board, or downloaded to a target transputer system.

3.7 Image Printing

A program was written to print out image data, stored in files on the host system, on a bit graphics printer. It was found that 16 grey levels could be produced on the printer, by different combinations of single graphic points, and different printer intensities. This gave a very good image printed on paper.

3.8 Summary.

A laboratory was equipped with image capture and display equipment, and transputer development facilities. The video equipment consisted of a 128 by 128 pixel CCD camera and digitiser, video monitor and monitor driver buffer. There was no framestore as such.

An Interface was designed and constructed using a link adapter, to allow a transputer to receive a complete image from the camera and digitiser, and to display an image on the video monitor.

The transputer development facilities included a M68000 based Stride and, subsequently, an IBM PC Compatible microcomputer. The Inmos Transputer Development System suite of software executed on an IMS B004 residing inside the IBM PC Compatible host.

A quad transputer board was designed and constructed, with several being plugged into a 19" rack for ease of use.

CHAPTER 4

QUAD TRANSPUTER BOARD HARDWARE

4.1 Introduction

Transputer Circuitry. Transputer based circuitry can be more straightforward to design than that using conventional microprocessors, due to the inherent design of the transputer. Circuits require a minimum of external components, as the processor is virtually self contained, and may execute with only an external clock [127]. The transputer can automatically generate all of the read and write, RAS, CAS, refresh strobes, and interface signals required to interface to external static or dynamic memory [25]. This dramatically reduces the overall chip count as compared with systems employing conventional processors [128, 129].

Rationale for In-House Construction. Quad-transputer boards were designed and built for use in the research project [130]. The main reason for in-house construction was financial, as commercial boards (Inmos IMS B003 for example) were very expensive when they were first introduced (1986). These commercial boards were also available after the first in-house custom prototype boards were produced.

Furthermore, most of the commercial boards, even at the time of writing, do not have all four links per transputer available at the edge connector. The ternary tree

configuration cannot be realised using IMS B003 boards, which have two links per transputer available at the edge connector. The design of the custom boards featured all four links per transputer available at the edge connector, for optimal flexibility and ease of use.

4.2 Circuit Design

External Memory Interface Configuration. The transputer has a semi intelligent External Memory Interface (EMI) which allows one of 13 preprogrammed timing configurations to be used. Alternatively each of the timing cycles can be set up individually, by using an external Programmable Read Only Memory (PROM).

The interface timing adopted used a predefined configuration selected by connecting the selection pin *MemConfig* to AD5, yielding the cycle states as shown in figure 4.1.

T1	T2	T3	T4	T5	T6	s1	s2	s3	s4	write	cycle type	e time
1	1	2	1	2	1	5	1	2	3	early	4	3

Figure 4.1. External Memory Interface Timing States.

Using these cycle states yields an interface timing diagram shown in figure 4.2.

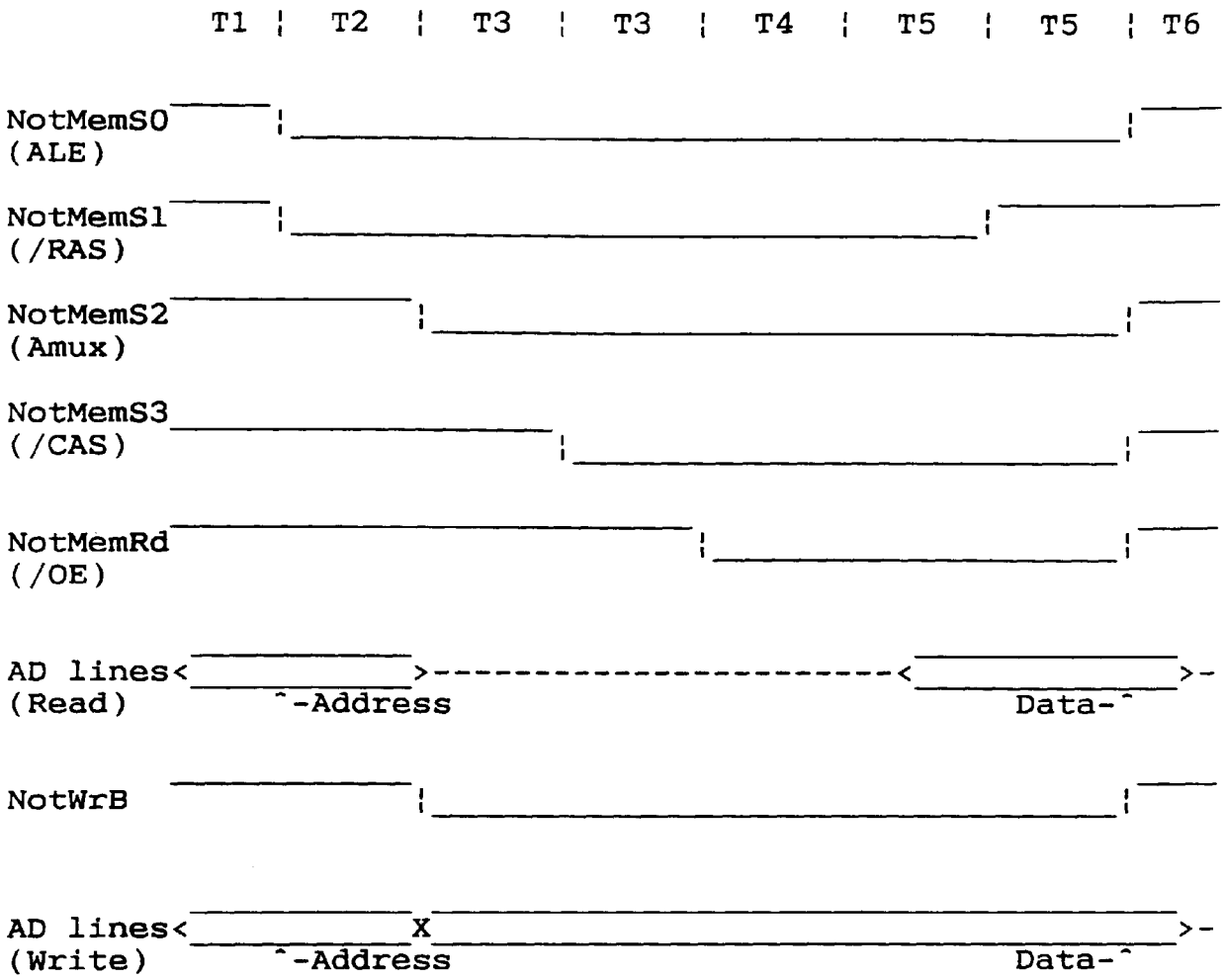


Figure 4.2. External Memory Interface Waveforms Used.

Data Routing to Memory Devices

As the transputer uses 32-bit words, the memory must be arranged in 32-bit wide addresses. As the memory device used was 64 Kbyte by four, eight such devices were required for each transputer memory block, arranged as shown in figure 4.3.

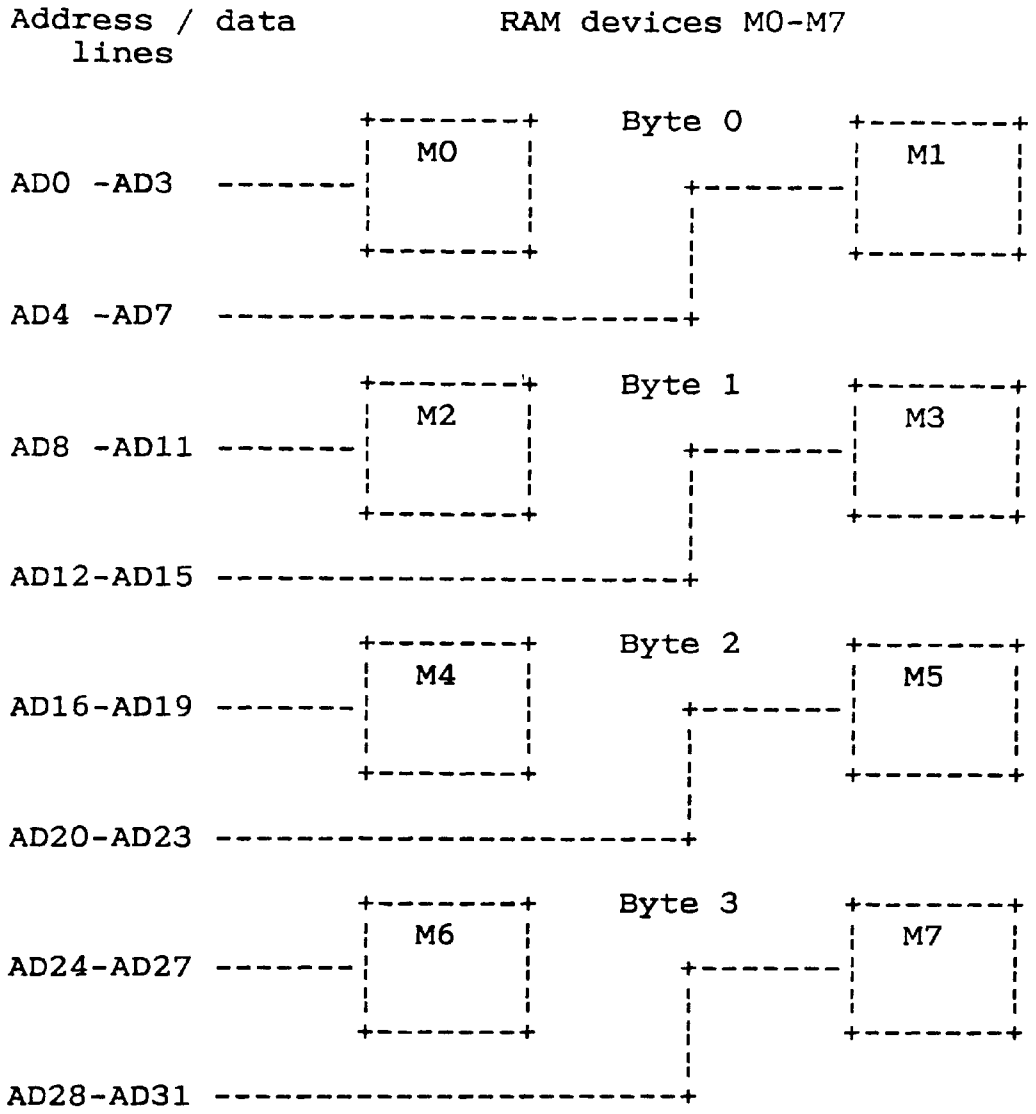


Figure 4.3. Data Routing to RAM Devices.

Transputer Block Design. The main part of the design of the transputer circuit block was associated with address and data demultiplexing. The transputer has a multiplexed 32-bit address and data bus, and as such, the address had to be latched and strobed into the DRAM in two stages, then data could either be read or written. Two multiplexors were used to select the low order address bits, while a latch held the high order address bits. After the first address strobe (/RAS), the latch was selected by the multiplexors, and the second address strobe occurred (/CAS). Then a data transfer could take place, depending

upon the state of the read or write signal. The outline of these multiplexors and the latch is shown in figure 4.4.

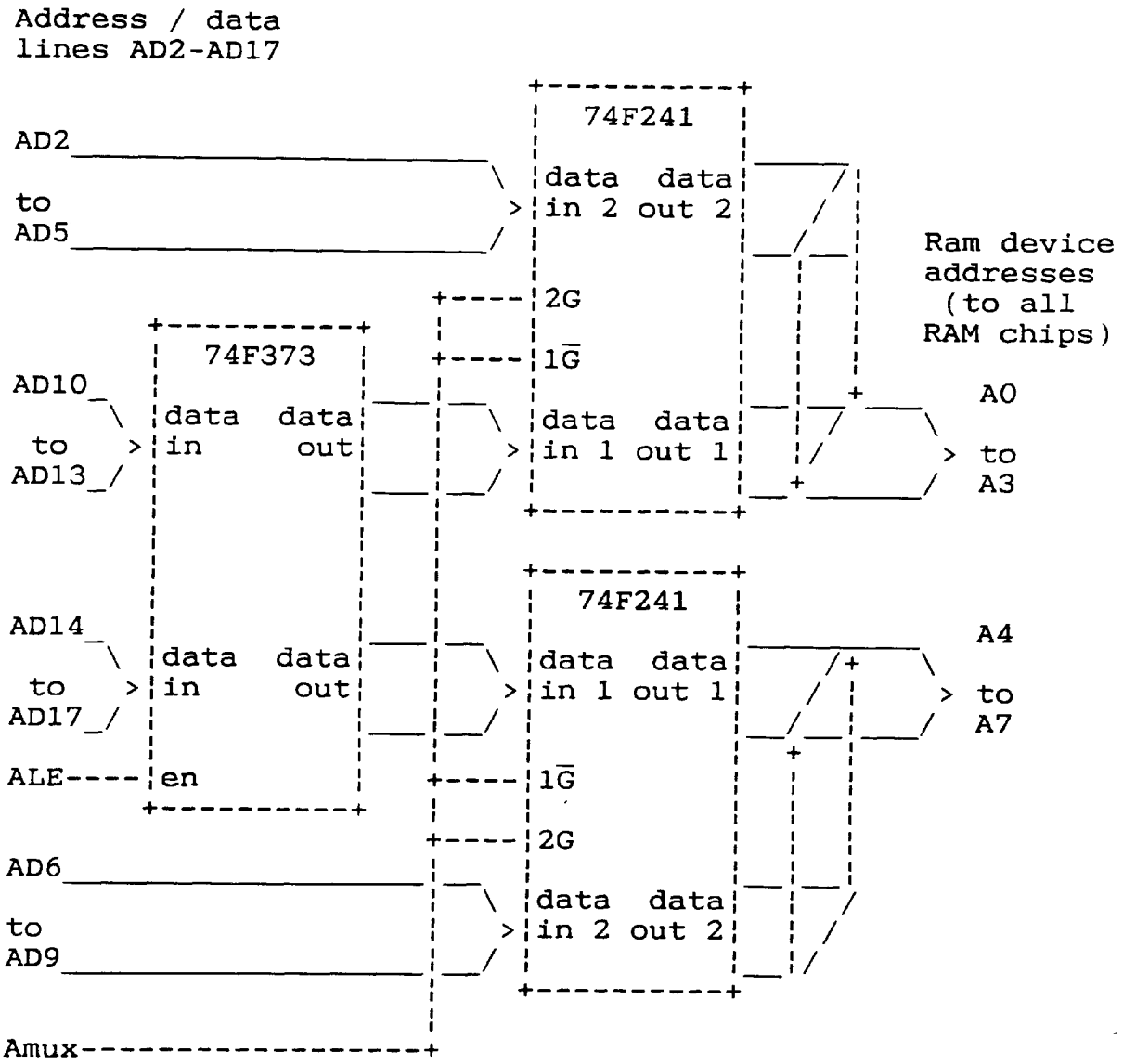


Figure 4.4. Address Demultiplexing, RAS and CAS address Generation.

4.3 Board Features

The basic specifications of the board were devised with respect to several factors, such as board size, component size, memory configuration, and cost. It was decided that a useful transputer arrangement would have a fairly large

memory, using the cheapest suitable devices. This determined that each transputer would have 256 Kbytes of dynamic RAM, using eight 64 Kbit by 4-bit devices, the 50464 or equivalent.

4.4 Board Construction

Each of the transputer blocks were identical (figure 4.5). A hand wired prototype board was constructed, and the circuit design proved, whereupon a second board was made. These hand wired boards did suffer a little from noise, especially when all four links were being used on all four transputers. This was reduced significantly by increased power supply decoupling on the board, but not eliminated completely.

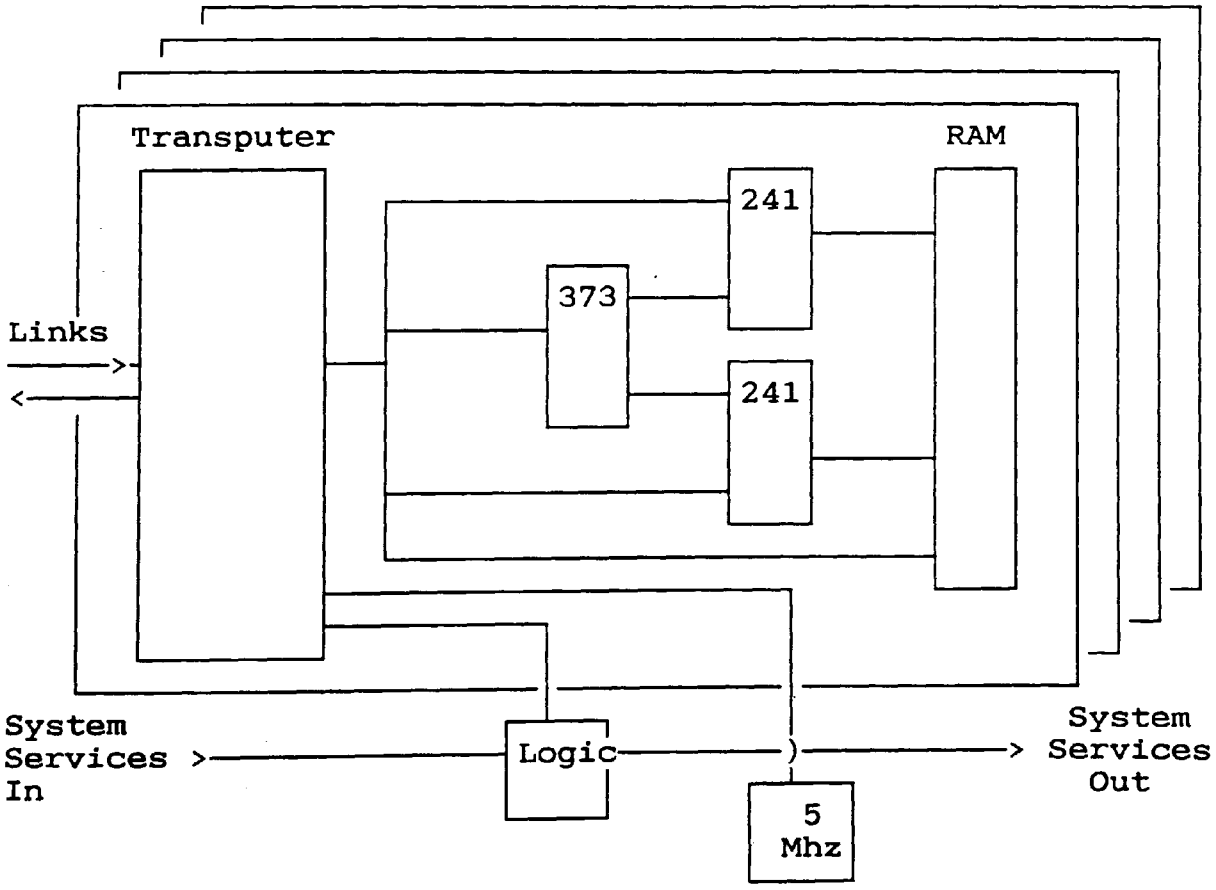


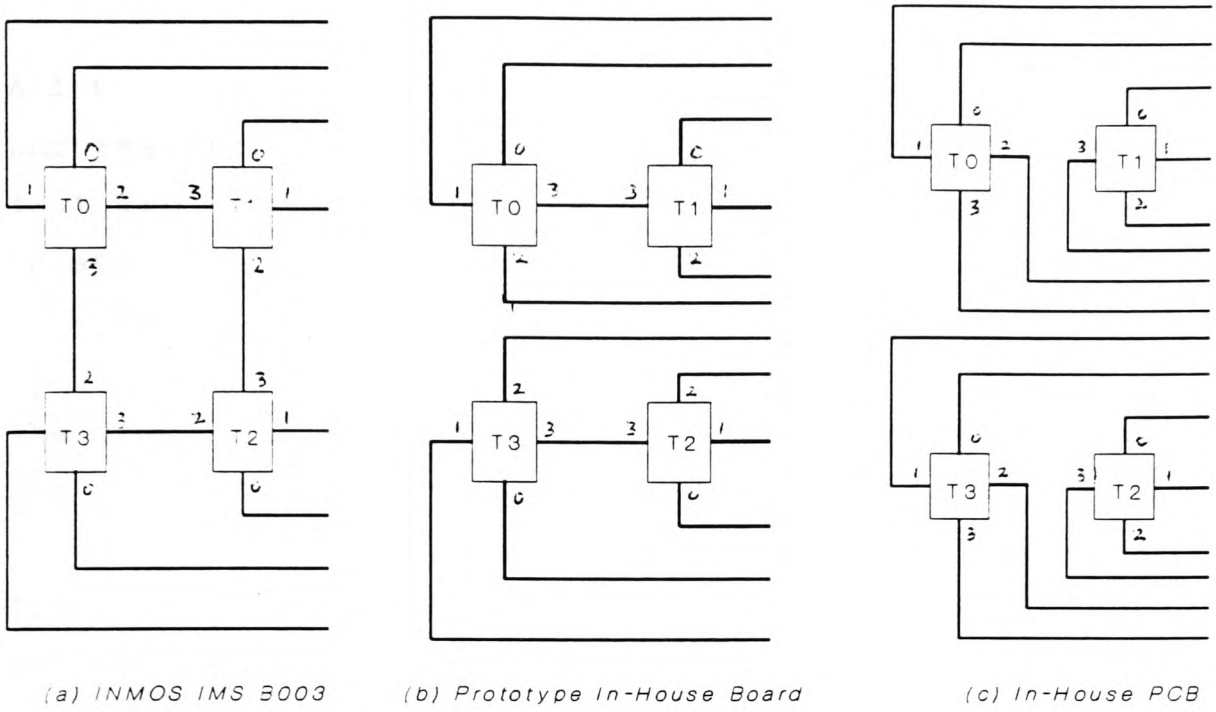
Figure 4.5. Quad Transputer Board Outline

When the finance was available, a four layer Printed Circuit Board (PCB) design was developed. The track layout was specified in detail by the author, and an external contractor optimised the overall layout using Computer Aided Design (CAD) facilities. Extensive checking was performed by the author on enlarged printouts of each of the layer artworks, to ensure that the correct tracks and connections were detailed. Laser produced negative photoplots were made, to act as the exact master artworks for the production of the boards.

The PCBs were then manufactured by a second external contractor, and constructed and tested in-house by the author. The PCBs were found to be completely faultless, and thus did not require any track deletions or wire adds whatsoever. They were then put into service. The noise problems were eliminated with these boards, as they had internal power and ground planes which made the two signal planes on either side of the board virtually isolated from noise pickup.

4.5 Transputer Links

It was realised that a ternary tree configuration could not be fully realised if less than 3 links per transputer were available at the edge connector. Some commercial boards, like the Inmos IMS B003 for example, only take 2 links per transputer to the edge connector, and have the other 2 connected forming a square arrangement with 2 of the other 3 processors on the board (figure 4.6a). It would not be possible to use such boards for configuring ternary trees.



(a) INMOS IMS B003 (b) Prototype In-House Board (c) In-House PCB

Figure 4.6. Transputer Board Link Connections.

When the first prototype boards were being used, with one hard wired link per transputer (figure 4.6b), it was often inconvenient to have to ensure that this hard wired link corresponded exactly to a channel between processors. This constraint meant that transputers could not always be used in a desired location in the network. For this reason, all 4 links were taken to the edge connector on the in-house PCB, for optimum flexibility and ease of use (figure 4.6c).

4.6 Extra Board Features

A few other features were incorporated on the PCB. Link speed select pins on the transputer were taken to bit switches so that 5, 10 or 20 MBPS link speeds could be easily set up. Some of the other pins designated *Hold To Ground* on the T414 are used to select options on the T800,

so these were connected to bit switches to allow T800 transputers to be used.

All the link inputs were terminated correctly, with a diode and resistor pair. The link outputs were buffered with fast TTL buffers, to ensure no problems would occur with noise, capacitance and inductance associated with off-board link cables.

The transputer external memory interface configuration was set up as a 4 cycle selection, but a facility was included to allow this to be changed to 5 or 6 cycle by cutting a track and inserting a short connection wire. This feature was included to enable the use of slower memory devices, or faster processors, such as a faster T414 or T800. Both options could possibly require an interface cycle with more processor cycles.

4.7 Testing and Debugging

The hand wired prototype boards were built in four sections, these being the individual transputer blocks. Each block was tested and validated before the next was constructed.

An extensive and thorough RAM test was implemented in occam, which wrote and read various test patterns to each RAM location. This was achieved using two methods. Initially, a second transputer was used to peek and poke values into and from the memory of the device under test.

A more comprehensive test consisted of an occam program executing on the transputer under test, exercising the RAM,

and links. The addresses of the RAM locations were also written to the RAM, to ensure that no address lines were incorrectly wired, or shorted together. Link tests were performed on the board as a whole, so that each processor was connected in a four node crossover ring, communicating to three others. The PCBs were tested in a similar manner.

4.8 Quad Transputer Board Specification

The specifications of the quad transputer board were very impressive, considering the small board area that was used, the low number of external devices required, and the clean circuit design. The board, when fully populated, had four 15 or 20 Mhz transputers, offering 30 or 40 MIPS total processing power. The processor used could be a T414 or T800, offering the option of having 6 - 8 MFLOPS.

Each transputer had 256 Kbytes of dynamic memory, with a fast 4 processor cycle interface configuration. The board thus had 1 Mbyte or DRAM in total. The link speeds for the processors were switch selectable to be 5, 10, or 20 MBPS. All four uncommitted links for each transputer were available at the edge connector, with terminated inputs and buffered outputs.

The board conformed to the INMOS standard of using three *Up* and three *Down* system control lines, for RESET, ERROR, & ANALYSE. All the board connections were available at one 96-way a,b,c edge connector, including power. This made the PCB easy to use with a rack housing.

4.9 Conclusions

A quad transputer board was designed and constructed. At the time of the board design (1986), limited literature was available for the layout and construction of multi-transputer boards, and the configuration of the memory. The hand wired prototype boards were actually being used within the research project before commercial equivalents were widely available. Since 1986, INMOS have published more extensive design details, which confirmed, and are in many respects similar to the in-house quad transputer boards.

The board was designed to use T414 or T800 processor types. Test programs were written to extensively test and validate each processor on the board.

Each transputer had 256 Kbytes of 4 processor cycle interface DRAM, and all four communication links available at the edge connector. This allowed optimal flexibility in the connections to other processors, and other boards.

Option pins on the transputers were taken to bit switches on the PCB, to enable easy selection of link speeds, and processor speed (with the T800).

The External Memory Interface could be selected to have 4, 5 or 6 cycles, by moving a jumper connection on the board. This not only enabled the board to be used with slower RAM, but also allowed a faster processor to be used, that might require a longer interface cycle.

The resulting extended double Eurocard sized board (10" square) had 30 - 40 Mips processing power (depending upon

the speed of the processor used), 1 Mbyte total DRAM, 16 links available at the edge connector. The board could be fitted into a rack, together with other boards, for convenient use.

CHAPTER 5

HARDWARE AND SOFTWARE CONFIGURATION CONSIDERATIONS

5.1 Introduction

In general, medium and high level image processing operations involve significantly more complex and hence lengthy computations than low level transformations. Any system that performs operations involving all three categories, such as an automated inspection cell, will therefore spend most of its time performing the more complicated algorithms. This aspect may be accentuated when dedicated VLSI devices perform low level transformations in real time. It therefore follows that the greatest gains in overall performance may be obtained by enhancing the execution performance of the complex operations.

5.2 Low Level Image Processing

The implementation of low, medium, and high levels of image processing operations has been investigated. Low level image processing transformations can be mapped onto a regular array very effectively. All sections of the image have to be processed in exactly the same way, regardless of the position of the image section within the image, or the value of the pixel data. This non data-dependence makes static load balancing possible and effective. The computation can be efficiently mapped onto an array of

processors, statically, before run time [131].

Arrays are thus ideally suited for these low level highly repetitive operations, where the transformation exhibits geometric parallelism over the image. Each processor works on that section of the image that corresponds to its position in the array. Neighbouring pixel values that are not held by a processor can be obtained from the corresponding neighbouring processor.

The computational load at each processor node is independent of the value of the data in the image being transformed. Only the operation type determines the workload of the task at each processor node. This will be almost constant over the entire image.

5.3 Higher Level Image Processing

The amount of computation required for medium and high level image processing algorithms, such as feature extraction and scene interpretation, is generally *entirely* data dependent. The amount of work necessary to extract features from some parts of an image may be negligible, whereas other parts of the same image may require a lot of work. This non deterministic data dependence can only be resolved dynamically at run time.

If a simple geometric division of work were implemented, as in the case of low level transformations on a processor array, and there was a large portion of an object of interest in a given part of the image, the respective processor would have a large amount of work to perform. It is likely that other processors would not have any

significant loading, if there was little or none of the object in their corresponding image section.

It is therefore apparent that the execution of these operations on a multi-processor configuration can only be increased using *dynamic* load balancing techniques. It is proposed that a *ternary tree* configuration is most suitable for the implementation of dynamic load balancing techniques. Work is not divided between the processors in any fixed predetermined way, being divided entirely at run time.

Processors that are given sections of the image that do not contain any features of interest will perform little, if any, processing, and will thus be allocated another section. Other processors that are given a image section containing complicated features may have to perform a large amount of work on it, and may only process that section. All the processors will typically process different numbers of image sections, depending upon the distribution of features of interest within the image.

There is not a great deal of advantage to be gained by this approach when dealing with non data-dependent operations, as the computation time is effectively constant from one processor to another, for a given image section size. Indeed, this arrangement would be less efficient than static workload distribution, as the low level data would have to be communicated around the system, whereas with a processor array, the data resides in the associated node.

5.4 Ternary Tree Advantages and Disadvantages

It was the major aim of this research project to investigate the implementation of image processing algorithms of various different complexities [132].

It was thought that a regular array would require significantly more complex software and greater inter transputer communications in order to achieve the desired dynamic load segmentation and distribution.

Advantages. A tree structure was therefore chosen to be investigated as the hardware configuration in this research project for several reasons :

1/ The structure has the *shortest distance* from the root node to any other node that can be achieved using standard transputer hardware. This is important, because both data and message transmission from the root node to other nodes, and vice versa, are more efficient due to the shorter path length. Not only do transmissions reach their destination in a shorter time, going through less intervening devices, but they also consume less processing resource of relaying devices.

A pipeline of *N* processors has a maximum distance from the controller node (at one end) of O(N-1), and from the controller node at the centre of the pipeline, of O(N/2).

With a square array of *N* four connected processors, such as transputers, the maximum distance from a control node (such as a corner node) to another node is $O(2 * N^{1/2})$.

With a binary tree of *N* processors, the maximum distance

from the root node to another node is $O(\log_2 N)$. Using a ternary tree reduces this value to $\log_3 N)$. Clearly, the number of links per transputer determines the logarithm base for this calculation, and hence the maximum distance.

Ideal *D* dimensional hypercube structures having *N* processors, and *D* connections at each node to other nodes would have a maximum distance from the controller node (one corner) to another node of D or $O(\log_2 N)$. This configuration cannot be realised above order four, however, using standard transputer devices and single transputer nodes.

2/ It is necessary to implement more complex image processing algorithms using dynamic load balancing techniques on a multi-processor configuration, in order to achieve satisfactory processor utilisation. A *processor farm* environment, where work is farmed out to processors, processed, then passed back to the controlling device, provides an ideal framework for this to be achieved.

The tree structure has a *high fan-out* capability to other processors from the root device. The root processor is connected to the maximum number of devices, and each of them are connected to the maximum number of processors, and so on. This makes the structure ideally suited for implementing processor farming. The pipeline has a fan out of unity, so that each processor is connected to only one further processor. This pipeline arrangement means that all the data for processor nodes further down the line must be passed through all the devices at the start of the line.

3/ The tree structure is *highly extendible*. Extra processors can be added to the configuration easily without

having to alter the software executing on it. Processors thus added can be arranged so that the overall structure remains balanced, evenly constructed and equally loaded. This ensures that the resultant system performance increase is distributed equally. Extra processors could be added to the system simply by changing a single parameter in the occam automatic configuration statement (described later in this Chapter).

4/ The software required for data and work distribution is *more straightforward* than with other configurations. Communications can only be either up or down the tree branches, which allows quite straightforward communications routing and control software.

5/ There is a degree of *fault tolerance* in the tree network. If a processor were to develop a fault, then the system software could reinitialise itself so that this was allowed for, and continue working. Clearly this does assume that the faulty node was not the root node, although this could be catered for if necessary, by implementing a double root node.

6/ As discussed in Chapter 1, it is highly likely that future transputer devices will have more on-chip communication links. This will allow higher order trees to be constructed, with the resulting performance advantages that the increased fan-out will allow. If the next transputer device has eight links, as projected in table 1.4, then order seven trees could be used (figure 5.1). This would reduce the maximum distance from the root node to another node to $O(\log_7 N)$, using N processors. The software devised and implemented by this research could be directly mapped onto these higher order trees.

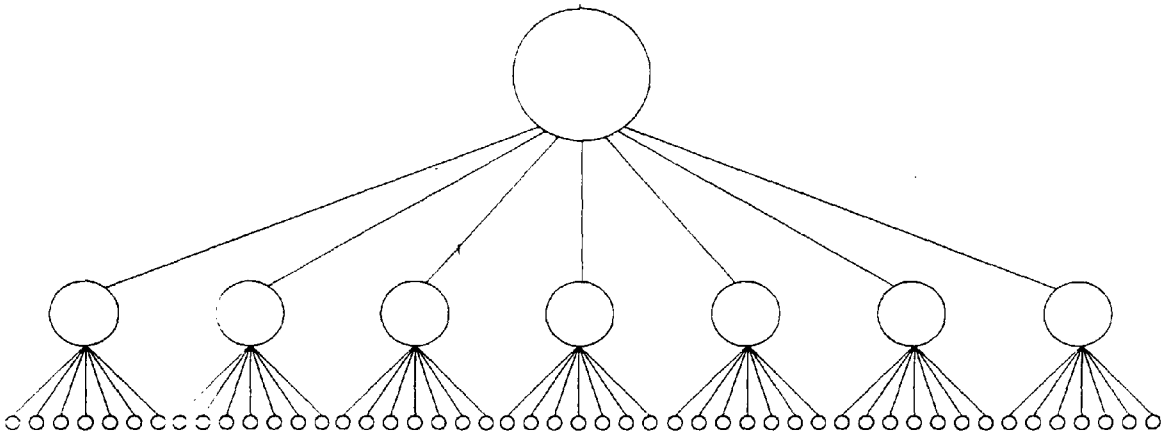


Figure 5.1 Order Seven Tree Architecture.

A ternary tree configuration has subsequently been studied and used by other research projects [99, 98], after some papers had been presented [134, 136, 147] detailing some of the research contained in this thesis.

Disadvantages. The tree configuration does have some disadvantages however.

1/ With low level image processing transformations, large amounts of data are being transmitted through the system, and the root node may become communication bound, where it is has a high level of activity on its links.

2/ If several operations were to be applied to the same data, using an array configuration, it is found to be particularly effective for the data to remain in each node, and the transformations applied sequentially. Communications between neighbouring processors could then exchange bordering values required for the subsequent operation. The image data would not have to be collected and redistributed in between each of the transformations.

With the tree architecture, processing on image sections that do not require access to neighbouring pixel values outside of that section can be carried out successively. Such operations are virtually self contained, in that data output from the first transformation is the data input to the next transformation. No neighbouring pixel values contained in other image sections, processed by other processor nodes, are required.

If the first transformation in such a sequence did require access to neighbouring pixel values outside the image section, these would have been supplied automatically by the system. Further operations that do not use neighbouring pixels can proceed as above.

Combining a sequence of operations that do require neighbouring pixel values leads to an inefficient implementation on the tree. This is because the image data has to be reassimilated and redistributed after each operation, due to pixel values on edges of image segments being changed by other processors.

3/ With larger tree configurations, some data to be transformed will be passed all the way to the bottom of the tree before it will be used. It will subsequently be passed all the way back up to the root node again. This is an inefficiency with the tree design, being increased in severity with lower order trees. It is not significant with small trees, but is more important with larger structures.

A situation could arise where the final image section to be distributed for transformation was passed to the bottom of a large tree, while nodes nearer the top were idle, waiting

for work. This could occur due to the data request queuing system implemented. This data could be operated on by a node at a higher level, in which case the transformed data would be ready quicker. This situation can be greatly alleviated by the use of streamlining or overlapping techniques, whereby nodes start the processing of operation *N+1* before all the data has been received by the root node from operation *N*.

5.5 The Order of the Tree

The Ideal Tree. The ideal tree that could be used for the implementation of the above mentioned techniques using *N* processors would be of class or order N-1. The root node would then be connected directly to *every* other node in the configuration, affording the maximum fan-out possible. This arrangement would mean that all communications in the system would be direct, between the root node and all other nodes. This would require a root node with (N - 1) + 1 = N communication channels, as one is required for the host computer connection.

The Realisable Tree. Transputers have a fixed number of communication channels or links, which is four at the time of writing, though it seems likely that this will be increased in the future [12], to eight (see table 1.4). This means that the highest order tree that can be readily realised at present is 4 - 1 = 3, or *ternary*. It would be possible to map IMS C012 link adapter devices [133] into the memory to increase the number of links available at each node. This option was not taken, however, because it would yield a non-standard solution, requiring custom hardware, and was thus outside the scope of this research

project.

5.6 Automatic Tree Configuration

The tree structure allows extra processors to be easily added. A configuration program (figure 5.2) was written in occam, that automatically configured an evenly balanced ternary tree structure, depending upon the number of processors available. Replicated PLACED PAR constructs were used to PLACE *software* processes onto the correct *hardware* transputers. The channels and links required for the configuration were all correctly placed and declared, and passed as actual parameters to the relevant procedures.

For any processor device *N* in the network there are four up.branch and four down.branch channels that must be placed on the correct four input and four output links. Each processor node had up.branch and down.branch channel suffixes going up the tree corresponding to its own identity.

```
PLACED PAR level = 1 FOR number.of.levels - 1          -- done level 0
  VAL number.required.at.this.level IS power.3 [ level ] :
  VAL start.of.this.level IS number.at.preceding.levels [ level ] :
  VAL number.available.for.this.level IS number.of.processors -
   number.at.preceding.levels [ level ] :
  VAL avail.negative IS (number.available.for.this.level /\ #80000000) >> 31:
  -- IS 1 for < 0 , IS 0 for >= 0
  VAL number.available.for.this.level IS number.available.for.this.level -
   (avail.negative * number.available.for.this.level ) :
  VAL diff IS number.available.for.this.level -
   number.required.at.this.level :
  --
  -- The smaller number out of "available..." and "required" is taken.
  -- this allows partially populated lower levels.
  --
  -- if diff > 0, then this level can be fully populated, using
  --  "number.required..." processors.
  -- if diff < 0, then this level cannot be fully populated, so must be
  --  partially populated, using "number.available..." processors
  -- if diff = 0, then either can be used, "number.required..." is used.
  --
  VAL req IS number.required.at.this.level :
  VAL avail IS number.available.for.this.level :
  VAL avail.wanted IS (diff /\ #80000000) >> 31 :  -- see if -ve
  VAL req.wanted IS (avail.wanted >< 1) :   -- mutually exclusive
  VAL number.at.this.level IS (avail.wanted * avail) + (req.wanted * req) :

  PLACED PAR machine = start.of.this.level FOR number.at.this.level
    PROCESSOR machine T4
      VAL up IS machine :
      VAL down.0 IS machine + number.required.at.this.level :
      VAL down.1 IS machine + (number.required.at.this.level * 2) :
      VAL down.2 IS machine + (number.required.at.this.level * 3):
      VAL table.ptr IS machine * 4 :
      PLACE down.tree.links [ up ]     AT link.lookup [ table.ptr ] + 4 :
      PLACE up.tree.links   [ up ]     AT link.lookup [ table.ptr ] :
      PLACE up.tree.links   [ down.0 ] AT link.lookup [ table.ptr + 1 ] + 4 :
      PLACE up.tree.links   [ down.1 ] AT link.lookup [ table.ptr + 2 ] + 4 :
      PLACE up.tree.links   [ down.2 ] AT link.lookup [ table.ptr + 3 ] + 4 :
      PLACE down.tree.links [ down.0 ] AT link.lookup [ table.ptr + 1 ] :
      PLACE down.tree.links [ down.1 ] AT link.lookup [ table.ptr + 2 ] :
      PLACE down.tree.links [ down.2 ] AT link.lookup [ table.ptr + 3 ] :
      demand1 (down.tree.links [ up ], up.tree.links [ up ],
               up.tree.links [ down.0 ], down.tree.links [ down.0 ],
               up.tree.links [ down.1 ], down.tree.links [ down.1 ],
               up.tree.links [ down.2 ], down.tree.links [ down.2 ],
               machine)
```

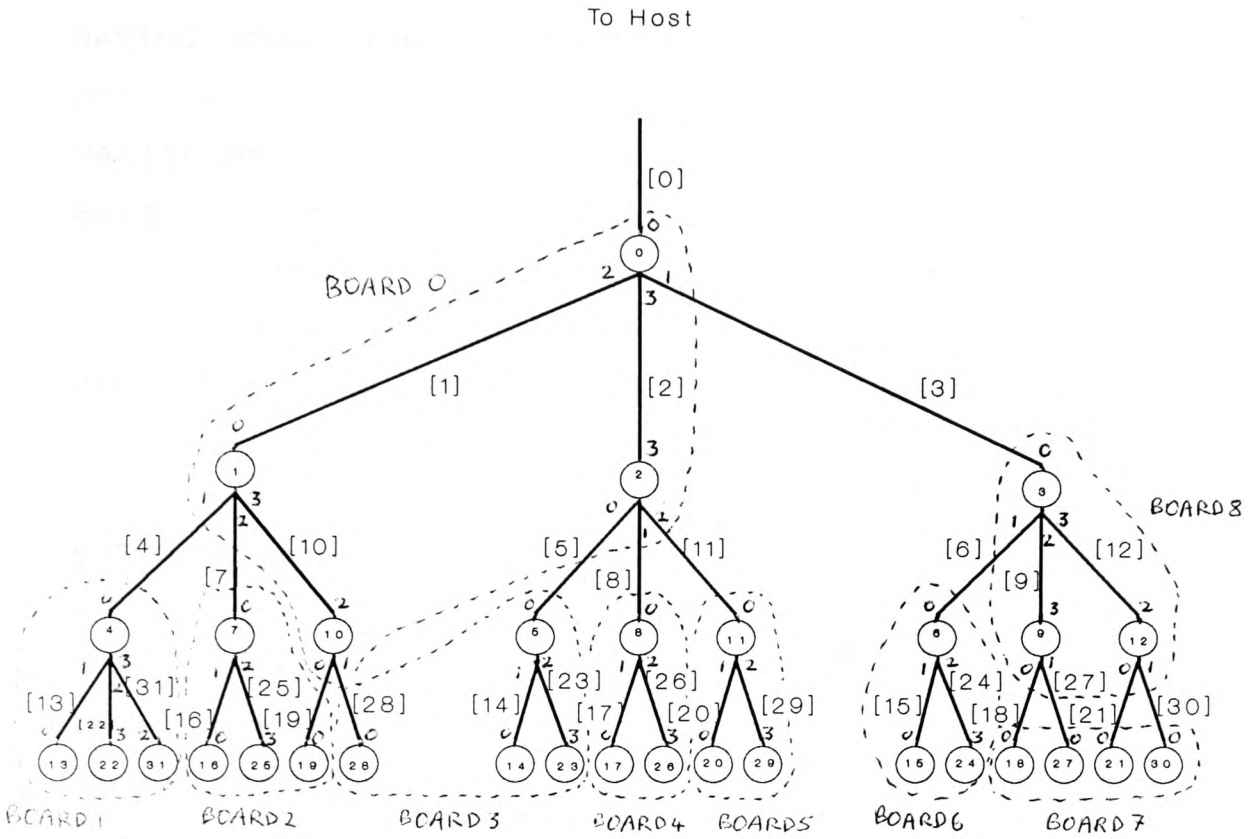
Figure 5.2 Automatic Configuration Program.

The automatic configuration program is shown in figure 5.2. This is the coding that places all channels and software processes in the tree, except the root node channels and process. The root process is placed by a similar section of coding. As can be seen, look up tables are used for powers of three, and for generating the total number of processors present at the preceding levels. This latter value was produced by summing the powers of three at each level in the tree, up to but not including the level being worked upon, and is required in order to determine the identity of the starting node at this level.

Each level in the tree is dealt with by a separate PLACED PAR construct, which allocates the software processes to the hardware processors, and places the channels onto hard links. Two channel vectors are declared, for the links connected up and down the tree.

Incomplete Final Level. It is interesting to note that the final level need not have its full complement of devices. This is allowed for in the program, with either the full number of processors that would be required for that level, or the number of processors left for that level, being taken as the replicator for the PLACED PAR.

Node and Channel Suffixes. Each node has channels connected to the next level up, suffixed as its own identity. The identities of the channels to connect to the next level down, therefore, must be determined by calculating the identities of the respective nodes that would logically be in the next level down. An example tree configuration is shown in figure 5.3. This network has 32 nodes, and was used successfully with an INMOS ITEM 400, in conjunction with the in-house quad transputer boards.



Note: Logical node numbers are shown inside the nodes. Logical channel numbers are denoted [*N*].

Figure 5.3 Example Tree Configuration with 32 Nodes.

When automatically configuring a tree with a certain number of processors, the program had to calculate the identities for processor nodes that were not actually being used. This occurred because the links of processors at the bottom of the tree were left floating, i.e. not connected at one end. It was necessary to place channels onto them, however, in order to achieve a standard procedural call. These channels corresponded to nodes that were not present in the network. There were always more channels placed in

the tree, therefore, than there were processor nodes.

Formulae Complications. The formulae were complicated by having to use the configuration subset of occam, which does not allow, for example, conditional operations and variables. Look up tables had to be used for evaluating some arithmetical expressions. Other problems were overcome where necessary by careful bit manipulations, shifting, and multiplication. As a result, the formulae are rather long and involved, but do achieve their aim.

5.7 Conclusions

Static load balancing is feasible and effective for low level image processing transformations. The non data dependency of such operations means that the computational load is almost constant throughout the image. An image can therefore be efficiently mapped onto a regular array, using geometric parallelism.

Higher level algorithms, having non-determinate computational requirements, cannot efficiently use the same strategy. Dynamic load balancing techniques must be employed, sharing out the work load at run time.

A Ternary Tree configuration of transputers is proposed, that allows the inherent high fan out to share the work load between the available processors. The tree network is highly extendible, working with any number of processors from 1 upwards. The structure proposed is found to be disadvantageous when used with low level transformations, however, but is effective for Feature Extraction and Scene Interpretation, where the data volume is kept to a minimum.

An automatic configuration program was written in occam, that correctly placed channels and processes onto and number of available processors. By changing a single parameter in the software, a different number of processor nodes could be used. The tree network has been successfully used with one processor, and with different numbers up to 32 processors.

CHAPTER 6

INTERACTIVE IMAGE PROCESSING FACILITY

6.1 Introduction

An interactive image processing system was designed and developed to execute on the multi-transputer network [134]. This was designed to be a skeletal framework within which widely varying image processing operations could be realised and executed. The framework facilitated the realisation of, and investigation into, new algorithms and techniques.

The interactive software was devised so that all the image processing operations implemented could be invoked from the keyboard, with any parametric control information being supplied by the user. Images could be captured and digitised from the camera, using the link adaptor interface [107]. Images could be operated on, and the transformations inspected both visually on the video display, and computationally using other image processing operations. Relevant feature information could then be passed back to the user on the host computer Visual Display Unit (VDU). The laboratory equipment setup is described in detail in Chapter 3.

The software framework provided an environment for the algorithms, performing the required "housekeeping", and facilitating the implementation of new techniques. Full option and parameter checking was performed on parameters

passed to the algorithms from the user. Image update and display were both performed automatically by the software framework. A timing facility could be switched on, to produce execution timings for algorithms.

6.2 Macros

A macro facility was developed, that enabled sequences of operations to be grouped together, and executed as a single burst by invoking the macro from the keyboard. This was a powerful tool, as once a sequence of operations was defined, a macro could be written that emulated a target application. The operation of the overall sequence could then be appreciated.

6.3 Registers

Parameters and results could be passed between operations, via registers, for producing more complex interrelated and data dependent algorithms. The values obtained using the feature extraction algorithms could also be stored in these registers for subsequent use. Simple arithmetical operations could be performed with the registers, using constants, and results from previous feature extraction stages. One example of this was in the computation of Shape Factors, which are calculated from the perimeter squared divided by the area.

6.4 Implementation of New Algorithms

The implementation of new algorithms was greatly facilitated by the software framework within which they executed. No external peripherals had to be allowed for, as this was all performed by the framework.

The action of an operation had to be initially specified, so that its effects and results were clearly defined. The algorithm to achieve the operation had to be designed, the implementation realised in occam, and its performance tested on appropriate test image data. The results could then highlight some aspects of the algorithm that may be corrected, improved or altered, in order to obtain a correct and efficient implementation.

The operation of low level image enhancement algorithms was independent of the image data values, as discussed previously, and did not therefore involve conditional execution or repetition of occam statements. They were not prone to unrecoverable errors which were sometimes exhibited by more complicated algorithms. The complex operations might deadlock, or exhibit infinite looping due to a software bug or unforeseen circumstance. This made the software debugging more difficult.

When higher level algorithms were to be implemented, however, it was necessary to analyse and design from first principles. All possible error or special case conditions that might result in erroneous operation had to be investigated and allowed for in the software. The interactive system itself assisted in the testing, debugging and validation of more complicated algorithms. The software framework facilitated the detailed testing of

these, by allowing the creation of special input image test data. Test data could be exactly tailored to that required, using other functions and facilities of the interactive image processing system.

6.5 Conclusions

An interactive image processing system was implemented that executed on a ternary tree of transputers using the SUPPLY and DEMAND software architecture described in Chapter 7. Single algorithms could be invoked, with specially created test image data, to investigate their operation and effects. Sequences of such algorithms could be grouped into Macros, and executed as a stream of commands. Registers enabled parameters to be passed between algorithms easily, and features extracted to be used in arithmetical operations. New algorithms could be devised, and tested, in order to ensure their correct function and robust execution.

CHAPTER 7

SUPPLY AND DEMAND SOFTWARE ARCHITECTURE

7.1 Introduction

Various requirements exist within a Multiple Instruction Multiple Data (MIMD) parallel processing environment that do not occur with traditional, wholly sequential implementations. The major issues are discussed below. A hardware and software architecture to solve many of these problems has been devised and implemented and is outlined. The software architecture was implemented using a ternary tree configuration, for reasons discussed in Chapter 5.

7.2 The Requirements for a Parallel Software Architecture

Communications. In a multi-transputer configuration, inter-transputer communications must be possible, for both data and message passing. There is typically no shared memory, although this has been implemented in some hardware projects [93]. This requires custom hardware, and was considered to be outside the scope of this research. Communicating sequential processes [135] executing on two directly connected transputers can communicate in the normal way via a serial link. The first process to become ready to communicate waits for the second to be ready before the data transfer takes place [25].

If two processes residing on two non-connected transputers require to communicate, then the normal synchronisation and

data transfer cannot take place directly between them. Some mechanism must exist in order that the message transmitted by the sending device can be routed to the correct destination processor via one or more intermediate devices.

Ideally some switching communications network would be connected to each transputer in the system, so that devices not directly connected may communicate via this external network. This would have the advantage of speed, as any message would be directly relayed onto its destination, and would not require significant message passing and routing software and processor resource in intervening devices.

It has been proposed by Inmos that this could easily be implemented within each transputer using dedicated silicon in the next generation of transputer devices [12]. Termed *Message Through Routing*, it would have the advantages of being semi-transparent to the processor, use minimal processor resource, would eliminate the software code required, and would be more efficient.

It was more convenient at the time of writing, however, for this mechanism to be implemented in software residing in each device that was required to perform data relaying. Each processor must have within it the capability of receiving data, inspecting it, and passing it on to the appropriate output link. The next device must behave in a similar fashion, either using the data itself, or relaying it on further to subsequent processors.
Ideally, a processor being used to relay data should be immediately available to receive data. This implies that the relaying mechanism would always be ready and able to input data, in order to introduce the minimum of delay to

the transmitting device. The entire message should be input and retransmitted without any significant delay.

Data communications are typically bidirectional, hence the relaying mechanism must be able to accept data input from any one of its links, and relay this to the correct output link.

Task Data Organisation and Segmentation. The computational task should be evenly divided between the available processors. This must be done in an efficient manner, so that devices are idle for as little time as possible, in order to have the task execution performance maximised. If the task division is uneven or inadequate, idle processors waiting for work will decrease the overall utilisation and hence the efficiency of the system.

Dynamic Load Balancing. Dynamic load balancing is essential in order that the execution performance of more complex algorithms is maximised. The software architecture must have the capability of supporting this function.

7.3 SUPPLY and DEMAND Features and Advantages

Inter-processor Communications. All inter-processor communications are either considered to be "down" or "up" branches of the tree. Communications originating at the root node and relayed down appropriate branches of the structure are termed *down*. Other messages starting at arbitrary nodes, being passed up branches to the root node, are termed *up*.
The root node controls the data division and distribution for operations being performed, and specifies the operation

to be performed on data. All communications in the configuration are either originated from, or directed to, this root node.

Work Distribution. All work is distributed from and by the root node. Each worker node has the ability to pass data down to other sub-nodes (or to itself), and to pass data up to its master node from sub-nodes (or from itself). In this way, data being transmitted down branches is passed to the relevant node. Similarly, data that has been produced by a node will be passed up the tree until it reaches the root. Each processor node can receive and transmit data concurrently, whilst continuing with its own parallel computation.

Dynamic Task Segmentation - Low Level Algorithms. All data segmentation and distribution is determined and performed *dynamically* at execution time. When dealing with low level non-data dependent transformations, the root node divides the input image data into many data sections. As computation associated with these operations only depends upon the operation being performed and the amount of data to be transformed, the system had the capability of dividing the data into groups of *unequal* sizes for distribution to available processors [136].

The use of unequal sized data sets simulated some degree of data dependency, in that computation times varied between processors in proportion to the data section size. This also alleviated the degree of the communications bottleneck at the root node, when all processors completed their computation at the same time.

Dynamic Task Segmentation - High Level Algorithms. The above solution, unequal sized data sectioning, was not required with higher level, more complex operations. The computation time for these depends upon the data values, not necessarily the data volume, and varies by several orders of magnitude. The approach used in this research utilised *chain coded object boundaries*. In order to operate on coded object edges, complete or partial chain code sequences were distributed. This technique had the advantage that the amount of data being passed around the system was dramatically reduced. The data was reduced from a two dimensional image section to a one dimensional sequence of numbers. This led to easier handling, and as the computation required on these sequences of numbers was highly data dependent, a far more effective workload scheduling system was achieved.

Load Balancing. Using the above mentioned data segmentation to achieve a more equal computational distribution, the workload was balanced more evenly around the transputers. Processor idle time was minimised, resulting in a more efficient overall implementation. As soon as an operation had been performed on some data within a node, the output data was passed to the root node, and the next packet of data was passed back. Hence processors given computationally intensive packets of data might only work on that one data section, while other processors may handle several, less computationally intensive data groups.

Extendibility to More Processors. Only a minor alteration was required to the configuration information for the software to take advantage of extra processors added to the system, as discussed in Chapter 5. The occam automatic configuration program ensured that the tree was extended

evenly, and that all the occam channels were placed correctly onto transputer links. Any number of processors could be used, from one to 32 or more, and in each case the tree was evenly balanced, and the software executing was exactly the same.

Optional DEMAND Process in Root Node. A DEMAND process could optionally execute in parallel with the SUPPLY process, in the root node (figure 7.1). This DEMAND process was run at low priority, so it only executed when the SUPPLY process was inactive, waiting for data from the external DEMAND processes. The DEMAND process would be descheduled almost immediately following an input to SUPPLY, so that the SUPPLY process's performance would not be adversely affected.

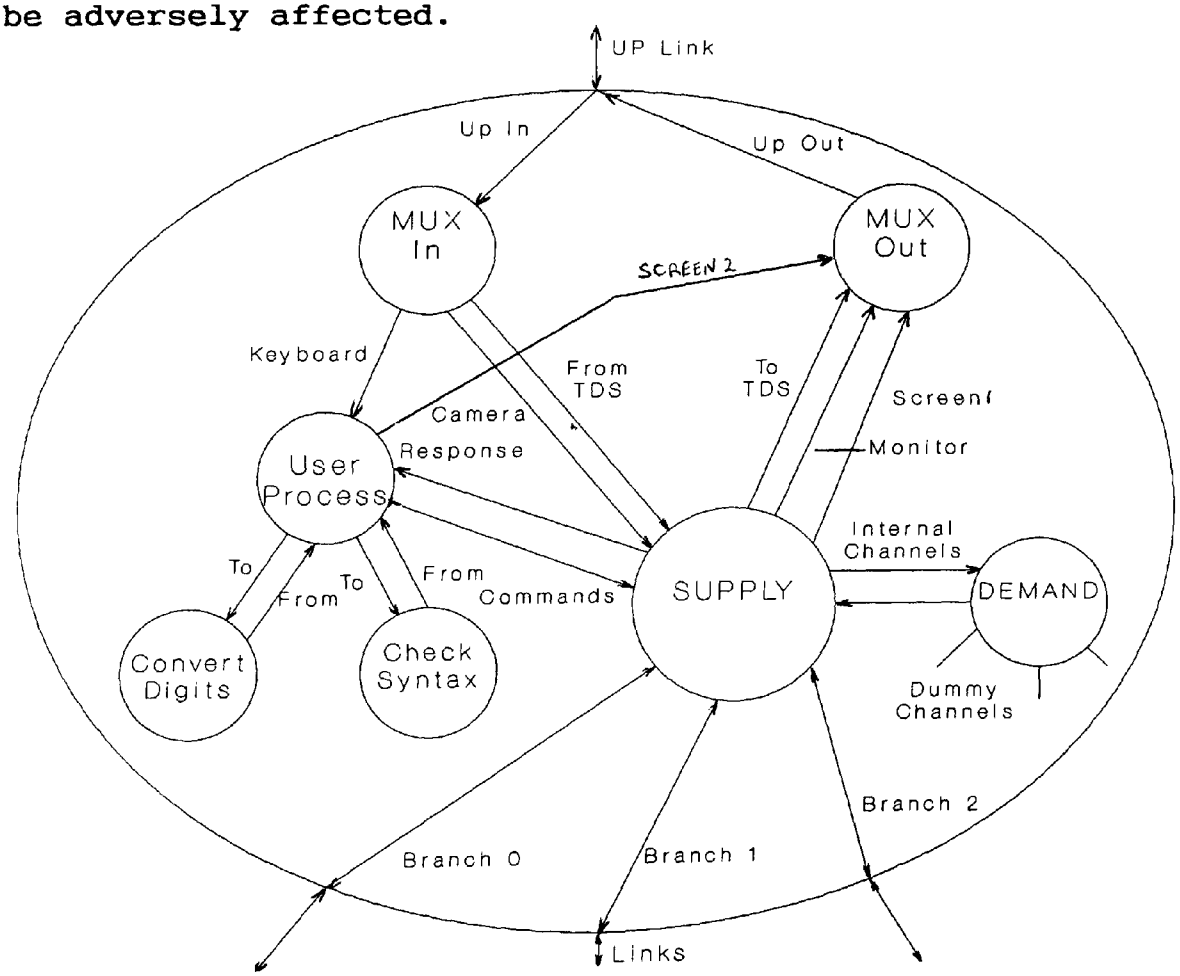


Figure 7.1. Root Node With Parallel DEMAND Process.

It has been found that this DEMAND process executing in parallel with SUPPLY did perform a significant amount of work, approaching 75% of that of an external DEMAND process, representing a useful contribution to the overall task execution.

7.4 SUPPLY

Function. SUPPLY executed within the root node, interfacing to the user program in the TDS, the link adaptor interface to the camera and display, and the tree network of transputer devices (and the optional internal parallel DEMAND process).

SUPPLY performed the central co-ordination of the work. Data to be transformed or operated upon is divided into sections, and transmitted to available processor nodes. Transformed data is then received back from nodes, placed into the correct relevant position in an array, and more data is transmitted to keep the DEMAND processes active, on a supply-on-demand basis. The operation that has to be performed on the data is selected, and changed according to the requirements.

Performance timings and efficiencies could be produced by SUPPLY, to enable the individual or overall operational characteristics to be examined. SUPPLY could also change the data group size dynamically when dealing with low level image processing algorithms, to determine the optimal size [136].

Initialisation. The initialisation section clears the image arrays used and performs other general housekeeping

tasks. The major part of initialisation, however, involves the execution of the SUPPLY part of the self-aligning algorithm. This initialisation in SUPPLY computes the number of DEMAND processes *N0*, *N1*, *N2* and *N3* connected to the internal channel, and each of the three branches *1*, *2*, and 3 of the tree respectively.

Obtaining the number of sub nodes on each branch enables the tree to be initialised more efficiently than would be achieved using the normal supply-on-demand technique. Processors cannot commence their computational work until they have received data to work on, so this "filling-up" operation must be performed as efficiently as possible.

```
PAR
  ...   transmit N0 data sections to internal branch
  ...   transmit N1 data sections to branch 1
  ...   transmit N2 data sections to branch 2
  ...   transmit N3 data sections to branch 3
```

Figure 7.2. Tree Initialisation.

The internal DEMAND and the three branches are initialised with data for transformation in parallel (figure 7.2). The first *N1* data groups are sequentially transmitted to branch 1, whilst at the same time, the next *N0*, *N2* and *N3* data groups are sent to the internal channel, and branches 2 and 3 respectively. If this more effective technique were not used, the resulting tree data initialisation would be more than *B* times longer, (where B=3, the number of branches) due to the internal DEMAND process.

The total number of sub nodes in the tree also determines the number of data groups that are likely to be required, in order to share the workload amongst the available processors. If there are *N* nodes, then it is unwise to

partition the task to be performed into less than N sections. Partitioning an image into at least 2 * N sections has been found to be effective.

Supply-On-Demand Operation. When the processor network has been initialised with data, SUPPLY waits to receive transformed data returned from the DEMAND processes. Whenever SUPPLY is inactive, the internal DEMAND performs useful work. Immediately data starts to be received, a new data section is transmitted to the appropriate tree branch *in parallel* with receiving the transformed data.

The reception of data from any one branch or channel is completely arbitrary, and is determined by the indeterminate operation of the DEMAND processes, and the generally uneven computational distribution. Thus one channel may be used in the communication of more data than another.

SUPPLY Outline - Version 1

```
SEQ
  ...   perform self alignment algorithm
  ...   initialise variables
  ...   initialise tree with data
  WHILE supply.on.demand
    SEQ
      ALT
        ...   start of input from branch 0
          ...   input and output data, branch 0
        ...   start of input from branch 1
          ...   input and output data, branch 1
        ...   start of input from branch 2
          ...   input and output data, branch 2
        ...   start of input from internal branch
          ...   input and output data, internal branch
  ...   get next command and loop
```

Figure 7.3. Supply Outline - Version 1.

Figure 7.3 outlines the first version of the SUPPLY process. The tree is initialised with data, then the supply-on-demand operational mode is adopted. All possible inputs are monitored, and an active one is selected, whereupon data is received from, and new data transmitted to, that branch, in parallel. The data received is placed in the appropriate location immediately it is input. This version of SUPPLY had a few limitations. Only one branch could be communicated with at any one time, instead of concurrently receiving and transmitting to all the branches that were active. Version 2 implemented this, using an ALT construct with a timeout guard, shown in figure 7.4.

```
INT start :
SEQ
  TIME ? start
  ALT
    input ? data
      ...   input rest of data, output data
    TIME ? AFTER start PLUS timeout.period
      ...   this channel not active
```

Figure 7.4. ALT Construct With a Timeout Guard

Version 2 of SUPPLY was able to concurrently input and output data from and to all four of the tree branches in parallel, as shown in figure 7.5. Each input was monitored within a short time, in parallel, so that any active branches were identified. This was achieved using an ALT construct with a timed guard, as in figure 7.4. All the details are not shown in the outline of version 2 of SUPPLY.

SUPPLY Outline - Version 2

```
SEQ
  ...   perform self alignment algorithm
  ...   initialise variables
  ...   initialise tree with data
  WHILE supply.on.demand
    SEQ
      PAR

        ALT
          ...   start of input from branch 0
            ...   input and output data, branch 0
          ...   no input from branch 0
            ...   do nothing

        ALT
          ...   start of input from branch 1
            ...   input and output data, branch 1
          ...   no input from branch 1
            ...   do nothing

        ALT
          ...   start of input from branch 2
            ...   input and output data, branch 2
          ...   no input from branch 2
            ...   do nothing

        ALT
          ...   start of input from internal branch
            ...   input and output data, internal branch
          ...   no input from internal branch
            ...   do nothing

      ...   copy data received to global image array
      ...   update variables

  ...   get next command and loop
```

Figure 7.5. SUPPLY Outline - Version 2.

The Value of the Timeout. The value of the constant *timeout.period* is important. If it is too long, then each of the branch inputs will be monitored for too long a period, resulting in a longer supply-on-demand loop cycle time. This cycle time is the delay incurred between inputting a data section and inputting another data section from the same or another branch, assuming that the branches are sufficiently active. Clearly this cycle time must be

as small as possible, to allow the fastest possible servicing of data from and to a branch.

Suppose that branch *0* had two data sections ready to be input to SUPPLY, and none of the other branches were active. If SUPPLY input the first data section, but spent a long time unproductively monitoring the other branches, then the second data section would not be input from branch 0 until the monitoring had timed out.

The timeout can also be too short, however, also resulting in an ineffective parallel communications implementation. It could happen that branch *0* became active, whereupon it was serviced with data, whilst in parallel, one or more of the other branches became active just after they had been timed out. This too short a timeout would then necessitate the other now active branches to wait until the next cycle of the supply-on-demand operation. This delay would be determined by the time difference between the timeout and the time taken to service branch 0. These two situations are illustrated in figures 7.6 and 7.7.

Case 1 - Timeout Too Long

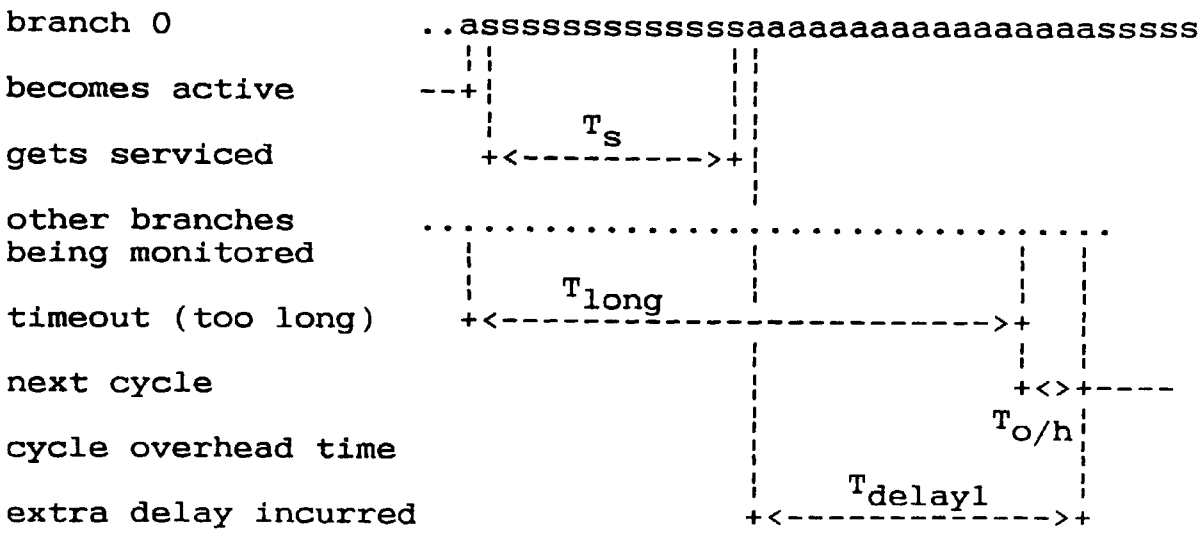


Figure 7.6. Too Long a Timeout.

Case 2 - Timeout Too Short

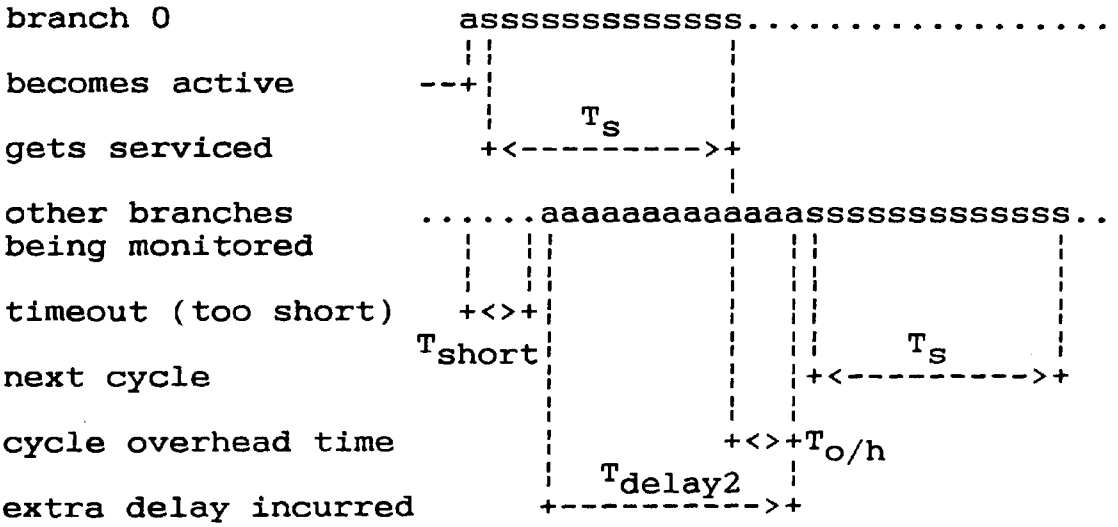


Figure 7.7. Too Short a Timeout.

In figures 7.6 and 7.7, *a* represents active, and *s* represents being serviced. T_s represents the time taken to service a branch, $T_{o/h}$ is the time between supply-on-demand cycles, and T_{long} and T_{short} are the long and short timeouts respectively. T_{delay1} and T_{delay2} are the resulting time delays incurred due to these values of the

timeout periods.

It can be seen that

$T_{delay1} := (T_{long} + T_{o/h}) - T_s$ equation (1)

and $T_{delay2} := (T_s + T_{o/h}) - T_{short}$ equation (2)

If we assume that for the minimum delay,

$T_{delay1} = T_{delay2}$ and that $T_{long} = T_{short}$

then from (1) = (2)

$T_{ideal} + T_{o/h} - T_s = T_s + T_{o/h} - T_{ideal}$

hence $T_{ideal} = T_s$

So that the ideal time T_{ideal} to monitor the other inputs would appear to be T_s, the period of time required to service an input. One further complication exists, however, to make this value non-ideal. At the worst case, one of the other inputs could become active at the end of the monitor timeout period, with branch *0* also being active with its second data section, as shown in figure 7.8. In this situation, the other branch would be serviced, thus adding T_s to the delay time T_{delay1}. If the monitor timeout was less than T_s, then these two branches could be serviced concurrently.

Case 3 - Timeout Too Long With Another Branch Active

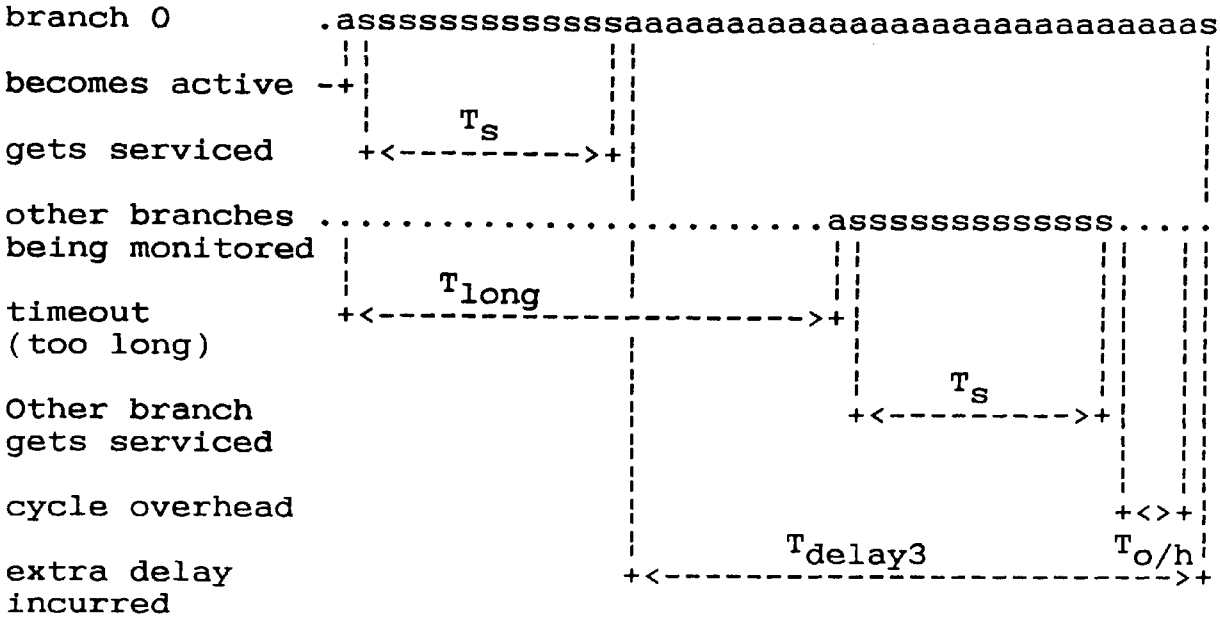


Figure 7.8. Too Long a Timeout With Another Branch Active.

$T_{delay3} = T_{long} + T_{o/h}$ equation (3)

equating equations (2) and (3) as before

$T_{ideal} + T_{o/h} = T_s + T_{o/h} - T_{ideal}$

hence $T_{ideal} = T_s / 2$

The ideal timeout value for SUPPLY is thus shown to be approximately half the time required to service an active branch. This represents an average figure, as the timing diagrams show the worst case scenarios.

7.5 DEMAND

Function. The DEMAND process provides the facilities for the communication of data to and from other processes or processors, and for the computation of output transformed

data. All communication is decoupled from the computational process.

Three parallel processes, *DOWN*, *COMPUTE*, and *UP* execute continuously within the DEMAND process (see figures 7.9 and 7.10). DOWN and UP handle communications down and up the tree, as well as data to, and data from, COMPUTE, respectively. The COMPUTE process performs the actual algorithmic work, being serviced with data by DOWN and UP.

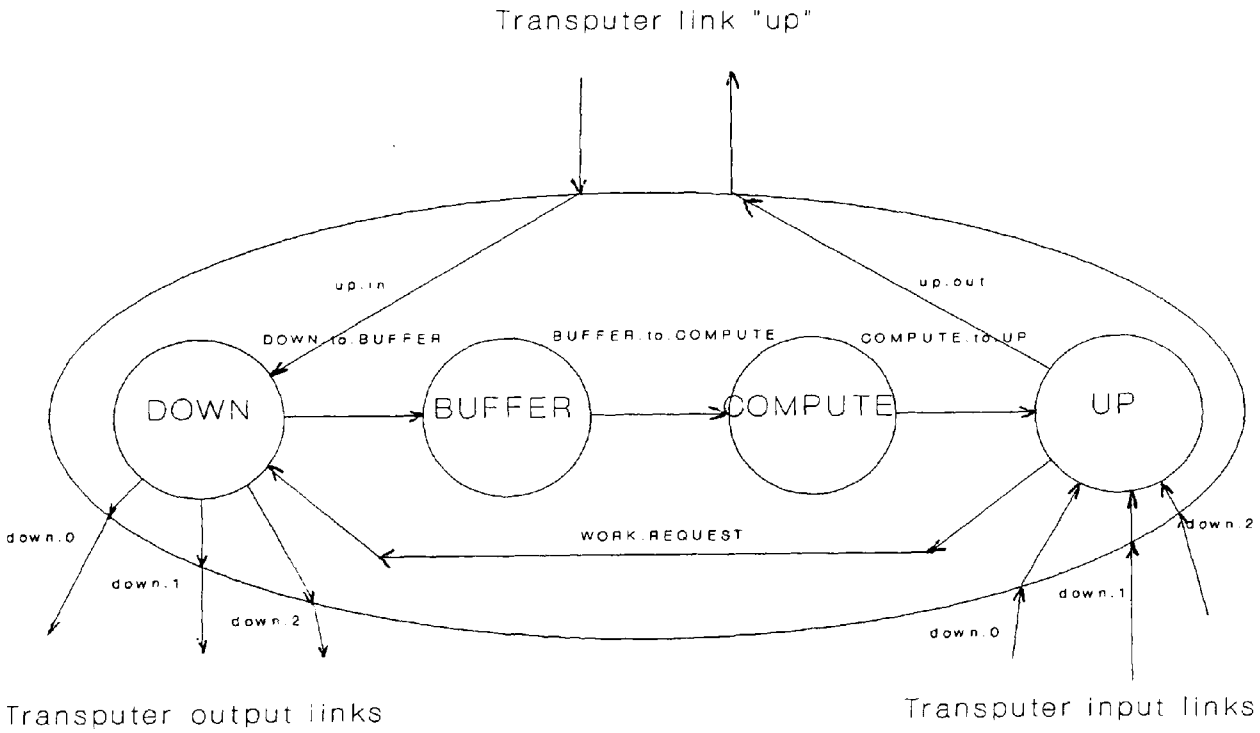


Figure 7.9. DEMAND Process.

Optional buffer processes can be placed in between DOWN and COMPUTE, and / or in between COMPUTE and UP, in order to keep a spare input data group for COMPUTE, and to decouple COMPUTE from UP. In this way, COMPUTE would never be kept waiting while UP serviced one of its link inputs.

DEMAND Process Outline

```
SEQ
  ...   initialisation
  PRI PAR
    PAR
      down( )
      up( )
      buffer.in( )
      buffer.out( )
    compute( )
```

Figure 7.10 DEMAND Process Outline

Initialisation. The first part of the DEMAND process, the initialisation, forms this stage of the tree self alignment mechanism, whereby the overall tree software adapts itself to the number of processor nodes connected.

The UP process then initialises the WORK.REQUEST queue in the DOWN process by signalling a dummy data transfer from the COMPUTE process, transmitting the INT.REQ Identity Tag. A "dummy" data transfer of zero length is then transmitted to the up channel, so that the WORK.REQUEST queue in the DEMAND process above is also initialised correctly. If these operations were not performed, the WORK.REQUEST queue would not be initialised according to the network connectivity, and DOWN would not be able to transmit data to the required place.

The zero length dummy data transfers are all passed to the root node, which receives the correct number per branch, and discards them. Note that these could form the basis for starting the supply-on-demand operation, with a data group being sent for each dummy transfer received.

Process Prioritisation. The two communications processes DOWN and UP are executed at *high* priority, and the COMPUTE process at *low* priority. Optional BUFFER processes are

executed at *high* priority. This arrangement ensures that the communications are serviced quickly, and not held up by the computation process. Typically communications take little time to initiate, whereupon the data is copied to or from memory using background Direct Memory Access (DMA) operations, which use very little processor time [25]. The computational process continues executing while the communication takes place.

DOWN. All communications down the tree are handled by DOWN processes. The reception and retransmission of data is carried out in parallel, as shown in figure 7.11. This ensures that the retransmission of data from the DOWN process does not delay the receiving of the next set of data to the DOWN process. The initialisation sets the boolean flags *GOT.SOME.DATA.1* and *GOT.SOME.DATA.2* which control the transmission of data.

```
SEQ
  ...   initialisation
  PAR
    handle.q()
    WHILE TRUE
      SEQ
        PAR
          receive(data1)     -- receive data 1
          transmit(data2)    -- transmit data 2
        PAR
          receive(data2)     -- receive data 2
          transmit(data1)    -- transmit data 1
```

Figure 7.11. Outline of DOWN process.

Data sets are referred to by reference and not by value, where possible [137], to eliminate the copying of data from one array to another. Arrays are instead passed as actual parameters to procedures, either for receiving data into, or transmitting data from. The data does not have to be copied from an input array after receiving it, to an output

array before the data can be retransmitted.

RECEIVE and TRANSMIT Processes within DOWN. The RECEIVE and TRANSMIT processes are used in DOWN and UP. The processes are not identical in each case, however, as they have subtly different functions to perform. In DOWN, the process RECEIVE accepts data from one input, and TRANSMIT subsequently outputs the data to one of four output channels, depending upon the next Identity in the WORK.REQUEST Queue.

WORK.REQUEST Queue. A queue is managed within the DOWN process to control data distribution (Figure 7.12). Whenever UP receives some transformed data from one of its four inputs, it sends the appropriate Identity tag to the HANDLE.QUEUE process via a WORK.REQUEST channel. These Identity tags are appended to the end of the queue.

```
SEQ
  ...   initialisation
  WHILE TRUE
    ALT
      Work.Request.Channel ? Identity
        ...   add Identity to end of queue
      Request.ID.Channel ? anything
        ...   Send ID from start of queue
```

Figure 7.12. WORK.REQUEST Queue Process outline.

When DOWN receives some data to be relayed on to one of the four output channels, it requests the next Identity from the queue, on a First-In First-Out (FIFO) basis, and sends the data to that output. This Identity is then removed from the queue.

The reception of Identity Tags via the WORK.REQUEST channel from UP, and the reception of data from the *up* channel by

the DOWN process must be asynchronous and independent. Many Identity Tags may be input from the WORK.REQUEST channel and appended to the queue before one data group is received from the *up* channel. The operation of this priority queuing technique avoids any form of process data starvation, where a process is bypassed and seldom (or never) receives new data.

It may be noted that towards the end of an image processing operation when the final data groups are being sent out, it would be possible for DOWN processes to pass a new data group to the bottom-most DEMAND. It may well be that a higher level DEMAND process had completed its work, and should for greater efficiency be receiving the data group rather than the one further away. This represents one of the aspects which prompted the Streamlining or Overlapping operational mode to be devised.

COMPUTE. The actual computational work is performed entirely within the COMPUTE process. New data to be transformed or operated upon is input from DOWN, and the resulting data is passed to UP (figure 7.13).

```
SEQ
  ...   initialisation
  WHILE TRUE
    SEQ
      ...   input data
      ...   perform operation on data
      ...   output data
```

Figure 7.13. COMPUTE Process Outline.

The image processing operation that is required to be performed on the data is controlled by an Option word located at the start of the data group. The Option word implies the type and format of the data. The data within

the group could comprise of a two dimensional image segment consisting of actual pixel values, or a chain coded edge sequence, area and perimeter information, or a feature vector. It must be unpacked into the correct format from the single byte slice form that it is transmitted in.

UP. All data communications up the tree are handled by UP processes (figure 7.14). As with the DOWN process, the reception and retransmission of data is carried out in parallel. Data must be accepted asynchronously and independently from either the internal channel from COMPUTE, or one of the external channels from another DEMAND process. Data is referred to by reference where possible, as with the DOWN process, and not by value. The initialisation sets the boolean flags *GOT.SOME.DATA.1* and *GOT.SOME.DATA.2* which control the transmission of data.

```
SEQ
  ...   initialisation
  WHILE TRUE
    SEQ
      PAR
        receive(data1)
        transmit(data2)
      PAR
        receive(data2)
        transmit(data1)
```

Figure 7.14. UP process outline.

Whenever some data is received, the appropriate identity tag is sent to DOWN, via the WORK.REQUEST channel, in parallel with the data being transmitted to the *up* channel.

RECEIVE and TRANSMIT Processes Within UP. In UP, the process RECEIVE accepts data from any one of four input channels, depending upon which one is ready, whilst in parallel, RECEIVE also sends the appropriate Identity to

the WORK.REQUEST Queue, in DOWN. The process TRANSMIT then outputs this data to the *up* channel.

7.6 Self Alignment Algorithm

Incorporated within the software architecture is the ability to make use of different numbers of processors in the configuration. Advantage can be taken of extra transputers connected, or allowance can be made for a smaller number of devices. Certain types of deadlocks are thus avoided, by eliminating the possibility of inputting from, or outputting to, non-existent processors or unconnected transputer links [138].

This is achieved by the SUPPLY process interacting with the DEMAND processes, after the network has been loaded with the software by the host system. Each DEMAND process identifies itself to the process in the next level up, as shown in figure 7.15. It also waits a certain time for any DEMAND processes in the next level down to identify themselves. Each DEMAND process then inputs the number of sub-nodes in the levels below if any identified themselves. One is added to the total, (to allow for itself) and the final value is output to the next process up the tree via the *up* channel.

In this manner, the number of sub-nodes is progressively incremented and passed up the tree, until SUPPLY receives the top level numbers, representing the total number of DEMAND processes on each branch of the tree (figure 7.16).

```
{{{  initialisation
SEQ
  PAR
    up ! dummy                                        -- I'm here

    {{{  monitor input channels for an input -- are you ?
    SEQ
      ...   initialise variables
      SEQ i = 0 FOR 3
        PRI ALT
          ALT
            branch.0 ? anything
              branch.0.active := TRUE
            branch.1 ? anything
              branch.1.active := TRUE
            branch.2 ? anything
              branch.2.active := TRUE
          ...   timeout this pass
    }}}

  {{{  If you're there, how many others have you got ?

  SEQ
    {{{  check branch.0

    IF
      branch.0.active
        branch.0 ? no.sub.nodes.0
      TRUE
        no.sub.nodes.0 := 0
    }}}
    ...   check branch.1
    ...   check branch.2

  {{{  plus 1 for me, that makes...

  total.on.all.three := no.sub.nodes.0 +
   (no.sub.nodes.0 + no.sub.nodes.2)
  up ! total.on.all.three + 1
  }}}
}}}
```

Figure 7.15. DEMAND Self Alignment.

SUPPLY Self Alignment

```
PROC are.you.there()
SEQ
  ...   initialise variables
  SEQ i = 0 FOR 4
    ALT

      {{{  input from branch 0

      branch.0 ? dummy
        SEQ
          branch.0 ? no.sub.nodes.0
          SEQ i = 0 FOR no.sub.nodes.0
            SEQ
              branch.0 ? dummy
      }}}

      ...   input from branch 1
      ...   input from branch 2
      ...   input from internal channel
      ...   Time out this pass
```

Figure 7.16. SUPPLY Self Alignment.

7.7 Operation Selection and Control

Each of the DEMAND processes have a complete set of all the image processing algorithms that have been defined, and can work in parallel. The particular image processing operation that is to be performed is selected by an Option word which is inserted at the start of the data group. This would be selected by the user, or would form part of a Macro sequence. This Option word also inherently defines the type and format of the data, as different image processing operations assume specific input data, and generate specific output data. As an example of this, the command "Chain Code Edges" assumes an input data format of N by M (two dimensional image data) and an output data format of Chain Code (a sequence of numbers representing the chains produced).

7.8 Self Optimisation of Operational Parameters

The SUPPLY and DEMAND software architecture had the ability to alter certain performance parameters, and monitor the overall resultant image computation efficiency [136]. This allowed the effects of different parameter values to be compared, and the optimal values documented and adopted for subsequent use. The main parameters that were useful to change were the data group size for low level image processing algorithms.

7.9 Conclusions

In a multi-processor environment where each processor is autonomous, various requirements exist that do not occur with single processor systems. The use of transputers with no shared memory avoids the need for the complex, time consuming, and restrictive overheads traditionally employed with parallel processing.

This approach does necessitate some other requirements, however, that do not occur with traditional sequential systems. Communications, the task workload and task data distribution, and load balancing must all be taken into account. These factors can markedly affect the overall system performance.

A software architecture has been described that uses SUPPLY and DEMAND software processes executing on a ternary tree configuration of transputers. Any number of processors could be used. An automatic configuration program was written that correctly placed channels and processes on the appropriate hardware links and transputers. As more

transputers were added to the network, the configuration program added them evenly to the bottom of the tree.

The software initialised itself, on power on or reset, to determine the number of DEMAND processes on each branch of the tree. This enabled the tree to be efficiently initialised with data at the start of operations.

The DEMAND process consisted of four parallel processes. Two communications processes, executed at high priority, a computation process that ran at low priority, and a buffer process that executed at high priority. A WORK.REQUEST queue ensured that data was only passed to devices present, in the correct order of requirement.

The software allowed the task workload to be divided evenly between available processors. The higher levels of image processing, being highly data dependent algorithms, require different amounts of computation over an image. Dynamic work allocation and load balancing techniques were devised and employed that divided the non-determinate task workload between the transputers at run time.

CHAPTER 8

IMPLEMENTATION OF LOW LEVEL IMAGE PROCESSING ALGORITHMS

8.1 Introduction

The implementation of a variety of image processing algorithms within the multi-transputer configuration was achieved using the SUPPLY and DEMAND architecture described in Chapter 7.

Algorithms of essentially three different categories have been considered, low level transformations, medium level feature extraction, and high level scene interpretation [139]. This Chapter discusses the implementation that was devised for low level transformations.

Low level functions, being non-data dependent, involve a highly repetitive operation at very high speed. Typically images must be processed in real time, at 25 images per second. With image sizes being 128 by 128, 256 by 256, 512 by 512 or greater, there is a large amount of data being processed by a highly repetitive process.

Some examples are included here, to demonstrate how common, low level image processing operations were implemented in occam. The examples shown in this Chapter are for the transformation of an image section from an image of size 128 by 128, which, using the software architecture featured in Chapter 7, allowed a multi-transputer configuration to

perform the operation in parallel. Each DEMAND process performed the relevant operation on each image section that it received.

Single and Multi-processor Implementations. Nearly all of the algorithms have been implemented in parallel, using up to 32 transputers. Similar program codings were also used to perform transformations using a single processor, for operation validation purposes, in which case the image section was considered to be the entire image. It was useful for software development, therefore, to be able to use similar occam code with the SUPPLY and DEMAND architecture, to allow the multi-processor system to be realised.

Monadic Transformations. As discussed in Chapter 1, there are essentially four classes of low level transformations. The parallel implementation of each of these classes is subtly different. Monadic transformations involving a point-by-point replication of an operation over the entire image are the most straightforward to implement. The sub-class of operations within the Monadic group are similar, except that these involve an initial stage, where parameters of the image are noted, affecting the actual transformation to be performed in the second stage. Both of these Monadic classes can be performed in parallel on a multi-transputer configuration by partitioning the image into sections, each of which are operated on in the same way.

Local Operators. The replication of a Local Operator transformation over an image differs from a standard Monadic operation in that the neighbouring pixels of each element are included in the computation. When partitioning

an image into sections, pixels on the edges of sections require neighbouring pixels to be included from other sections. The extra pixel values are included for information only, as new values are only produced for pixels that are within each section. New values for the extra pixels included with section *N*, say, would therefore be computed within the neighbouring section *M*, where section N is adjacent to section M in the image.

Dyadic Operations. The implementation of Dyadic operations basically differs from Monadic only in that the transformation involves pixel data values from two images. Both images must be partitioned into sections, and one section is produced.

Image Transforms. The implementation of Image transforms differs from that associated with Monadic, Dyadic and Local Operators. An image transform involves a transformation that operates on the image as a whole, rather than a operation replicated on each pixel element. Geometric transforms that were implemented included axis inversion and halving, image shifting and rotation, and row maximum. These were not implemented in parallel, however, as it was not considered worthwhile within the research.

8.2 Data Division

The image was divided up into sections for distribution. Theoretically any practical shape sections could have been used for the operations that did not need neighbouring pixel values, as only the image data that was required was present. The sections had to fit together well, though, to allow efficient image sectioning, which implied a regular

shape section.

With transformations that require neighbouring pixel intensities, the shape of the image section used is more important. For each section that is distributed, an extra border of pixels from adjacent sections must also be included, to enable the transformation to take place over the complete section. Such extra pixels represent overhead, and the amount of overhead introduced is proportional to the perimeter of the image section. Hence section shapes that have a small perimeter for a given area will result in a smaller border overhead than shapes with a large perimeter.

For a given regular shape, the perimeter can be calculated in terms of its area. With a circle of area *A*, the perimeter *P* can be found :

$A = pi * r^2$ Equation (1)

$P = 2 * pi * r$ Equation (2)

Where r is the radius.

Eliminating r in (1) and (2)

$A = pi * (P / (2 * pi))^2$

Hence $P = 2 * (pi * A)^{1/2}$

or $P = 3.55 * (A)^{1/2}$

A circle has the smallest perimeter for a given area, therefore, being $3.55 * (A)^{1/2}$. For a given area of image data, the least number of extra bordering pixels will have to be included to distribute the image section. Circular sections would not be very practical, however, due to their inefficient packing. Large spaces would be left in between the shapes, leading to complicated extra sectioning to

process these, possibly involving overlapping shapes, and unnecessary duplicated processing.

The next most effective shape is that approximating a circle, i.e. that of an polygon. Octagonal and hexagonal shapes were considered, but discarded due to their complex packing and shape. For a square of area *A*, and perimeter *P*:

$A = s^2$ Equation (3)

$P = 4 * s$ Equation (4)

Where *s* is the side length.

Eliminating *s* in (3) and (4)

$A = (P / 4)^2$

Hence $P = 4 * (A)^{1/2}$

A square shape offers an acceptable perimeter for a given area, being $4 * (A)^{1/2}$. The square shape allows an efficient packing arrangement to be used. It is interesting to note that the perimeter of a square of area *A* is only 1.13 times that of a circle with the same area.

It was thought that a rectangular shape would offer a good compromise between the shape compactness of a square, and the handling of image arrays. All memory copying and communication is performed using array or memory slices on a transputer. Each slice must be assigned individually, so a data section that has fewer longer slices will be handled more effectively than one with more, shorter slices.

In the system, rectangular sections are used of varying sizes. The size of the section directly affects the degree of parallelism granularity that is obtained. If the

sections used are too large, the granularity would be too high, and the workload will not be sufficiently divided between the processors. If the section size is too small, the granularity would be too low, and there would be too much communicating of small image sections around the network. In addition, with neighbouring pixels being transmitted, these overheads would increase dramatically. In either case, the optimal data division would not be obtained.

If an image section is of size *N* by *M*, then for a 3 by 3 local operator, sections of (N + 2) by (M + 2) must be distributed. Thus the overhead introduced to allow for the neighbouring pixel values is

$$(N + 2) * (M + 2) - N * M = N * M + 2M + 2N + 4 - N * M$$

$$= 2 (N + M + 2)$$

The percentage overhead is thus

$$\frac{2 (N + M + 2)}{N * M} * 100 \%$$

Transformed data in sections of N by M were returned to SUPPLY. The number of overhead pixels to be distributed associated with a square image section of side N is given by:

$$\text{overhead pixels}_{\text{square}} := 4 (N + 1)$$

and with a rectangular section of sides N and M, is given by:

$$\text{overhead pixels}_{\text{rectangle}} := 2\ (N + M + 2)$$

These formulae include the four corner pixels as overhead. Table 8.1 illustrates the overheads involved with neighbouring pixels included, for different square and rectangular shaped image sections.

Image section size	No. of pixels	No. of Overhead pixels	Overhead Percentage
128 * 128	16384	516	3.1
64 * 128	8192	388	4.7
64 * 64	4096	260	6.3
32 * 64	2048	196	9.6
32 * 32	1024	132	12.9
16 * 32	512	100	19.5
16 * 16	256	68	26.6
8 * 16	128	52	40.6
8 * 8	64	36	56.3
4 * 8	32	28	87.5
4 * 4	16	20	125.0
2 * 4	8	16	200.0
2 * 2	4	12	300.0
1 * 2	2	10	500.0
1 * 1	1	8	800.0

Note : Rectangular section side dimensions are listed in conventional vertical, horizontal order.

Table 8.1. Rectangular and Square Image Sections.

From table 8.1, it can be seen that with larger square and rectangular sections, the percentage of overhead pixels is quite small, but with small sections, the overhead percentage can be very large. It has been found that a data group size of around 32 by 32 to 16 by 16 is suitable, in particular, a rectangular section of size 16 * 32 has been used successfully. The use of sections of these sizes allows the image to be divided into enough sections to enable the workload to be distributed evenly, without

incurring a large overhead due to the extra pixels required.

8.3 Data Group Distribution

With operations that do not require local neighbouring pixel values, two dimensional image data sections of size *N* by *M* are copied into a single dimensional array, within SUPPLY. The Option word that specifies the command to be performed is also copied into the array, together with the section size parameters, N and M, and any parameters required by the operation. This single dimensional array is then sent to one of the tree branches. DOWN processes ensure that the array is forwarded to the correct DEMAND process, as discussed in Chapter 7.

For image operations that do require neighbouring pixel values (from adjacent sections) to be included with the image data sections, procedures were written that include the appropriate pixel values. When such a section was being copied into the single dimensional array, the appropriate neighbouring values were included. Special cases occur at the image edges, where neighbouring pixels do not exist. The strategy adopted here was to use the image edge pixel values. This gave the advantage that false edges were not generated by the image edges, as would be the case if the non existent pixels were assumed to be any other value (such as black or white). It may be noted that corner pixels associated with sections lying on the image edge do not need to be transmitted, as their values are already included.

The single dimensional array is forwarded by DEMAND

processes to the appropriate COMPUTE process, where the data is unpacked and reformed into a two dimensional image section. The COMPUTE process subsequently computes the transformation, working on the image section, producing output data into another array. The output array is packed into a single dimensional array, as with the SUPPLY process, and passed to the UP process, for further transmission to the SUPPLY process.

8.4 Negate.

One of the most simple image processing operations is that of Negate. All the pixel values in an image are negated, so that black becomes white, white becomes black, and other grey levels are effectively subtracted from 255. This was implemented in occam (figure 8.1).

```
SEQ y = 0 FOR no.of.rows
  SEQ x = 0 FOR no.of.cols
    B(x,y) := A(x,y) >< #FF
```

Where *A(x,y)* is the input image section, and *B(x,y)* is the output, transformed image section.

Figure 8.1 Example of Negate.

This was implemented in parallel, using the SUPPLY and DEMAND software architecture.

Very low level image processing transformations, such as monadic operations, where pixels are replaced by some function of their value, may be performed using simple logic and pre-stored look up tables [3]. The speed obtained using this technique may easily be altered to suit the application, and real time performance is readily

available. Some logic may be fitted directly to a camera, to form a semi-intelligent camera unit, yielding real time thresholded binary images, for example.

A look up table may be used to implement Negate, as shown in figure 8.2, although with such a simple operation, the benefits of this are marginal.

```
SEQ y = 0 FOR no.of.rows
  SEQ x = 0 FOR no.of.cols
    B(x,y) := look.up.table [ A(x,y) ]
```

Figure 8.2. Negate Using a Look Up Table.

Results for the Negate Transformation. Tables 8.2, 8.3 and 8.4 show the performance obtained for the Negate transformation. Image sections of sizes 16 by 32, 16 by 16 and 8 by 16 were used, with an image of size 128 by 128.

Number of Transputers	Execution Timing/mS	Performance Speed Up	Efficiency
1	181	1	100%
4	58	3.12	78%
10	50	3.62	36.2%
20	57	3.18	15.9%
32	61	2.97	9.3%

Table 8.2. Negate Performance, using 16 by 32 Data Section.

Number of Transputers	Execution Timing/mS	Performance Speed Up	Efficiency
1	260	1	100%
4	79	3.3	82.5%
10	49	5.3	53%
20	42	6.19	31%
32	45	5.78	18.1%

Table 8.3. Negate Performance, using 16 by 16, Image Section.

Number of Transputers	Execution Timing/mS	Performance Speed Up	Efficiency
1	304	1	100%
4	89	3.4	85%
10	59	5.15	51.5%
20	57	5.33	26.7%
32	60	5.07	15.8%

Table 8.4. Negate Performance, using 8 by 16 Image Section.

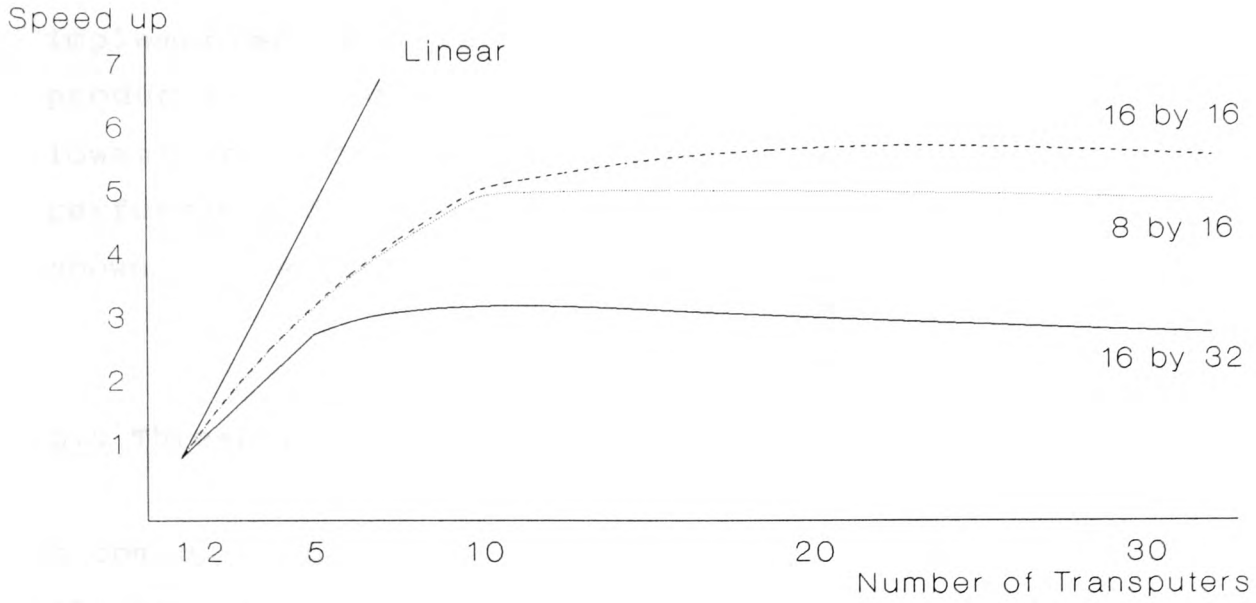


Figure 8.3. Negate Speed up Against Number of Transputers.

Discussion of Results. As can be seen from the graph (figure 8.3), the resultant speed ups for Negate are far from linear. The performance using a 16 by 16 image section is better than that obtained using a 16 by 32 section. The reason for this is that the granularity is reduced, allowing a more even division of work between the available processors.

The graphs indicate that the speed ups are approximately 70% of linear with up to 4 transputers. This is the main turn point on the graphs, as this is when the third layer of processors starts to be used in the tree. The efficiency of the tree markedly declines as more layers are added with this transformation. A second turning point on the graphs can be seen at about 13 transputers. This is when the fourth layer of processors starts to be used in the tree.

The Negate transformation has the lowest computational requirements of the image processing algorithms implemented, and works on two dimensional image data, producing two dimensional data. It therefore has the lowest work to data ratio, which accounts for the severe performance degradation using more than 4 transputers, as shown on the graph.

8.5 Thresholding.

A common image processing operation is that of Threshold. All the grey level pixels in an image that are equal to or above a certain precalculated or defined threshold value are set to white, and those below this value are set to black. This can be expressed in occam, as shown in figure 8.4.

```
SEQ y = 0 FOR no.of.rows
  SEQ x = 0 FOR no.of.cols
    IF
      A(x,y) < threshold.value
        B(x,y) := black
      else
        B(x,y) := white
```

Figure 8.4. Example of Thresholding.

A simple look up table transformation, such as thresholding, may be expressed in occam as shown in figure 8.5, where *no.of.intensities* is the number of discrete intensity values possible for the pixel representation being used (an 8-bit pixel representation allows 256 different intensities).

```
[ no.of.intensities ] INT look.up.table :
SEQ
  -- calculate look up table
  SEQ i = 0 FOR threshold.value
    look.up.table [ i ] := black
  SEQ i = 0 FOR (no.of.intensities - threshold.value)
    look.up.table [ i ] := white

  -- use look up table
  SEQ y = 0 FOR no.of.rows
    SEQ x = 0 FOR no.of.cols
      B(x,y) := look.up.table [ A(x,y) ]
```

Figure 8.5. Improved Thresholding using a Look Up Table.

It is interesting to compare this realisation with that obtained using a conventional sequential approach, as shown in figure 8.4. In the first realisation, (no.of.rows * no.of.cols) arithmetic comparisons and assignments are performed. In the second one, (threshold.value) + (no.of.rows * no.of.cols) assignments are made. The second version of this thresholding contains simpler instructions, and though it also contains indirect addressing, using an array element as the index to a table, it executed significantly faster than the first version.

The results for Thresholding were found to be essentially the same as for Negate, shown in tables 8.2, 8.3 and 8.4, and the graph in figure 8.3. This is not surprising, because although Thresholding used a dynamic look up table, the computation of the data for the table was not significant, and thus did not significantly affect the timings obtained.

Percentage Thresholding. An algorithm was implemented to allow a Percentage Threshold to be applied to an image. A number could be supplied that specified the percentage of pixels that were to be set to black (or white). Initially the algorithm had to compute a frequency table, that

contained the frequency of occurrence of each intensity. Using this table, the required threshold value was determined, to yield the specified percentage. The Threshold algorithm shown in figure 8.5 was then applied to the image data, using the determined threshold value.

Automatic Thresholding. An automatic Thresholding algorithm was implemented that calculated the threshold value depending upon the image data [140]. This proved to be most useful, as in most cases the threshold value that was produced was extremely close to the optimal value, found manually.

When this algorithm was implemented on the multi-transputer network, two different approaches were contrasted. The first required local parameters to be calculated within DEMAND processes, passed back to SUPPLY, and global parameters determined. The *global* threshold value was subsequently transmitted with the original data, for a simple Thresholding transformation to be applied, using the calculated global threshold value.

The second approach yielded a more efficient implementation. Each DEMAND process calculated its own local parameters from the image section. A Threshold transformation was then applied to the image section, using a *local* threshold value determined from the locally calculated parameters. This approach eliminated the requirement for an additional distribution of data, and consequently the algorithm proved to be dramatically faster than the first approach.

8.6 General Monadic Transformations.

Some common Monadic Transformations, such as Square, Double or Halve Intensities, can be performed in the straightforward fashion, in occam :

```
SEQ y = 0 FOR no.of.rows
  SEQ x = 0 FOR no.of.cols
    B(x,y) := F[ A(x,y) ]
```

Where F[ ] denotes the function being performed.

Figure 8.6. Common Monadic Transformation.

The technique shown in figure 8.6 is very inefficient, because the function F[ ] is being performed for each pixel in the image section, i.e. (no.of.rows * no.of.cols) times. Assuming that there are more than 256 pixels to be transformed in the image section (which would generally be the case), then a 256 element table can be calculated in advance, yielding a look up table for each pixel intensity [141]. To transform a pixel, its value is used to address the output value, from the table.

The table can be calculated at run time, for non-determinate functions, such as Contrast Enhance. With simple example transformations given above, Square, Double, and Halve, it is found to be more efficient to provide the look up table as a predetermined data table. Using a predetermined look up table for a simple transformation was implemented as shown in figure 8.7.

```
SEQ y = 0 FOR no.of.rows
  SEQ x = 0 FOR no.of.cols
    B(x,y) := look.up.table[ A(x,y) ]
```

Figure 8.7 Using a Predetermined Look Up Table.

If the data for the table is non-determinate, then a dynamically created table at run time must be used (figure 8.8).

```
-- create look up table
SEQ int = 0 FOR 256
  look.up.table [ int ] := F[ int ]

-- use look up table
SEQ y = 0 FOR no.of.rows
  SEQ x = 0 FOR no.of.cols
    B(x,y) := look.up.table[ A(x,y) ]
```

Figure 8.8. Using a Dynamic Look Up Table.

It is interesting to note that the predetermined look up table technique works well, regardless of the number of pixels in the image section. If an image section was very small, for some reason, having less than 256 pixels, and the look up table could not be predetermined, then it was found that this yielded a less efficient algorithm than that which was obtained using the straightforward technique. This was considered to be an exceptional case, however, as in all the examples considered here, the image section size is rarely below 256 pixels.

Look Up Table Shortcut. With some transformations employing look up tables, the function does not need to be calculated for all of the pixel intensities. Because output intensity values must be limited to the range *black* to *white*, intensities for which the calculated function is clearly outside this range can simply be set to black or white, as appropriate. As a simple example, consider the look up table data for the function Square Intensities. Clearly, all pixel values up to the square root of 256 must be calculated, but all intensities equal to and above this value are limited to 255, so do not need to be calculated.

8.7 Enhancement

The intensity spread of typical images obtained from a real scene tends to be predominantly around the darker levels. Most of the pixels in the image will have intensity values lower than the midpoint intensity level. This results in information and details not being clearly visible, as the contrast or intensity spread is not ideal. There are several methods for improving this situation, two of which have been implemented, Contrast Enhance and Histogram Equalisation. These techniques seek to redistribute intensity levels to other intensity levels, so that the overall intensity range is improved, often resulting in a dramatic improvement in the image contrast.

Contrast Enhance using a Dynamic Look Up Table. Contrast Enhance is a transformation where the pixel intensities present in an image section that lie within a certain range are adjusted so that they lie within a second specified range [18]. The most common form of Contrast Enhance adjusts all the intensities in a section to lie within the range *black* to *white*, so that the highest value would be white, and the lowest value black. The adjustment to each pixel intensity is quite complex, typically involving subtracting a shift value, multiplying and dividing by calculated parameters, then adding a second shift value.

Clearly this algorithm can make use of a look up table technique. The required values for each intensity within the input intensity range must be computed, then the pixel values can be used to address the output transformed value. The data for the look up table cannot be defined at compile time, however, as it depends upon the lowest and highest intensities present. A multiplication factor must be

calculated from these, as well as the shift values and a divider.

In figure 8.9, pixels with intensities within the input intensity range bounded by *src.lo* and *src.hi* are to be adjusted, to lie within the range bounded by *tgt.lo* and *tgt.hi*.

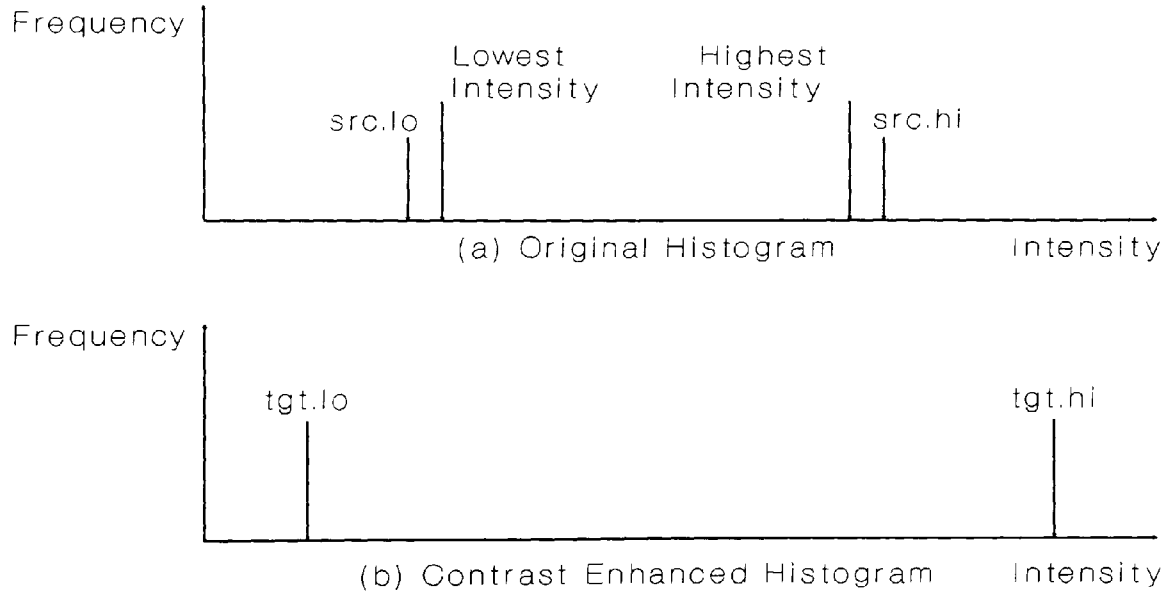


Figure 8.9. Contrast Enhance Intensity Ranges.

Referring to figure 8.9, a shift value of src.lo must be subtracted from each intensity being transformed. A multiplier of (tgt.hi - tgt.lo) and a divider of (src.hi - src.lo) must be applied, then a shift value of tgt.lo must be added, in order to scale the range src.lo to src.hi to the range tgt.lo to tgt.hi. In occam this can be achieved using a look up table (figure 8.10).

```
SEQ
  ...   obtain src.lo and src.hi
  shift.value1 := src.lo
  shift.value2 := tgt.lo
  multi.factor := tgt.hi - tgt.lo
  div.factor := src.hi - src.lo

  SEQ int = 0 FOR 256
  IF
    (int < src.lo) OR (int > src.hi) -- check range
      answer[ int ] := int
    TRUE
      SEQ
        new.value := ((int - shift.value1) *
         mult.factor) / div.factor
        new.value := new.value + shift.value2
        IF
          new.value > white
            look.up.table [ int ] := white
          new.value < black
            look.up.table [ int ] := black
          TRUE
            look.up.table [ int ] := new.value

  SEQ y = 0 FOR no.of.rows
    SEQ x = 0 FOR no.of.cols
      B(x,y) := look.up.table[ A(x,y) ]
```

Figure 8.10. Contrast Enhance Using a Dynamic Look Up Table.

8.8 Histogram Equalisation

The histogram is a graphical representation of intensity frequency against intensity. Histogram Equalisation modifies the image section intensities depending upon their frequency. The histogram must be modified, so that the image section pixels are distributed across the intensity axis, resulting in each frequency being set to the average frequency, given by :

Average frequency := no. of pixels / no. of grey levels

Histogram Equalisation Using A Dynamic Look Up Table. The average frequency can be calculated prior to run time, and provided as a constant. The ideal histogram equalisation would thus yield a uniformly flat histogram shape, resulting in the ideal intensity distribution. This is rarely possible to achieve, however, because the pixels' frequencies are generally found to be unequal at all the intensities. While it is possible to map two intensities to one, during intensity transformation, it is not possible to split one single intensity into two. For this reason, an approximation to a histogram equalisation must be made. An averaging technique must be employed, so that pixels are grouped together to give the approximate average level of frequency, along the intensity axis.

Algorithm. Pixels in each of the original image quantisation bands must be grouped together, starting at the lowest level, until the total is closest to the average frequency. These pixel intensity levels must then be set to the mid point of the enhanced image quantisation band. The process is then repeated for the next original image quantisation bands, until all pixels intensity levels have been mapped to a new level. In general, the number of discrete intensity levels in the enhanced image is less than in the original image, due to the grouping of levels. This effectively increases the quantisation error, but does dramatically improve the image contrast.

As with Contrast Enhancement, Histogram Equalisation must also use a dynamic look up table technique, as the intensity adjustment factors depend upon the lowest and highest values of the image data [18]. Contrast Enhance used a formula to determine each of the transformed pixel values, but Histogram Equalisation cannot, as the

intensities are mapped non-linearly into the output intensity range, depending upon the frequency of each intensity in the input image section.

Implementation. Figure 8.8 shows the outline of a Histogram Equalisation algorithm, in occam. A frequency array is constructed, which holds the number of pixels at each intensity. The average level for the histogram is calculated, given in the formula above. It was convenient to use three parallel processes to perform this algorithm, SENDER, FORMATTER, and RECEIVER. These execute as shown in figure 7.9. SENDER sequentially transmits each intensity frequency to FORMATTER. The FORMATTER process receives each frequency, and groups them together until the average level is exceeded, then passes the group on to the RECEIVER process. These groups are received by the RECEIVER process, along with the original intensity corresponding to the final frequency grouped by FORMATTER. A look up table is constructed, by RECEIVER, to translate the original intensities into the transformed Histogram Equalised intensities. Finally, the RECEIVER process must perform a certain amount of checking, to ensure that each original frequency has the correct new frequency in the look up table, due to the grouping of the original intensities.

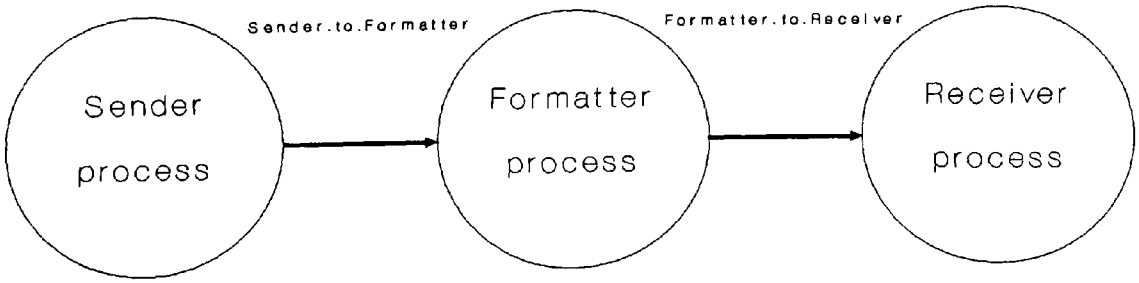


Figure 8.11. Histogram Equalisation Parallel Processes.

```
SEQ
  ...   initialise variables
  ...   build frequency array
  average.level := no.of.pixels / no.of.grey.levels
  CHAN OF ANY s.to.f, f.to.r :
  PAR
    SEQ                                    -- Sender
      ...  send out each frequency
    SEQ
      ...  receive and group frequencies, into
           (average value) sized groups
    SEQ
      ...  receive grouped frequencies, org intensities
      ...  check all elements of the look up table, to
           ensure all original intensities are correctly
           mapped to the new intensities.

  ...  perform look up table operation on all pixels in image
       section.
```

Figure 8.12. Outline of Histogram Equalisation Algorithm.

Parallel Implementation. In the parallel implementation, each processor builds a local frequency array, then communicates this to the root SUPPLY process. A global frequency array is then computed, by adding together the individual local arrays, then the global array can be sent to each DEMAND process, in order to perform the correct Histogram enhancement on each image section.

8.9 General Convolution

A convolution transformation may be employed to apply a spatial filter to an image. A convolution mask is passed over the entire image or image section, and the convolution operation is performed on each pixel. Each pixel is replaced by the sum of the products of the mask weightings and the appropriate neighbouring pixel values.

High speed convolution may be decomposed and performed in

parallel using a systolic convolver array [4]. Large convolution masks can be partitioned into smaller masks, to allow more accurate filtering.

Convolutions can be performed at high speed using dedicated hardware (figure 2.1). Delay units allow pixel and weighting factors to be multiplied and summed, in different combinations, to achieve the effect of moving the weighting matrix over the image.

If the mask weightings are contained within the array m[][] (figure 8.13), then a convolution operation may be expressed in occam as shown in figure 8.14.

0,0	0,1	0,2
1,0	1,1	1,2
2,0	2,1	2,2

Figure 8.13. Convolution Mask Array.

```
SEQ y = 0 FOR no.of.rows
  SEQ x = 0 FOR no.of.cols
    SEQ
      sum := 0
      SEQ m0 = -1 FOR 3
        SEQ m1 = -1 FOR 3
          sum := sum + ( A(x-m0,y-m1) * m[m1+1,m0+1] )
      B(x,y) := sum
```

Figure 8.14. General Convolution Outline.

Algorithm. In this straightforward occam realisation (figure 8.14), two SEQ loops are performed at each pixel,

to multiply the neighbouring pixel values by the weighting mask values. While this may be the most concise form of the algorithm, it was not found to be the fastest. As most of the work is performed by the two SEQ loops for each pixel, it was more efficient to unroll them. Loop unrolling can be a powerful method of speeding up the execution of a program. The individual parts of the loops were written out in longhand, so that the replicated overhead of the loop control mechanism was eliminated. The same Convolution algorithm with the two pixel loops unrolled is shown in figure 8.15.

```
SEQ y = 0 FOR no.of.rows
  SEQ x = 0 FOR no.of.cols
    B(x,y) := [ A(x-1, y-1) * m[0,0] + A(x, y-1) * m[0,1] +
                A(x+1, y-1) * m[0,2] + A(x-1, y) * m[1,0] +
                A(x, y)     * m[1,1] + A(x+1, y) * m[1,2] +
                A(x-1, y+1) * m[2,0] + A(x, y+1) * m[2,1] +
                A(x+1, y+1) * m[2,2] ]
```

Figure 8.15. General Convolution With Inner Loops Unrolled.

Notice that, in figure 8.15, unrolling the loops also eliminated the need for the extra variable, *sum*, significantly reducing the amount of variable accessing performed. The weighting matrix is also addressed by small constants, and not variables, which results in a more efficient direct address generation.

8.10 Laplacian Convolution.

To highlight edges, the Laplacian high pass filter using a 3 by 3 convolution mask may be used. This utilises a mask or weighting matrix defined as :

```
-1   -1   -1
-1    8   -1
-1   -1   -1
```

The values in the weighting matrix allowed a simpler and faster version of algorithm than was obtained using the general case convolution. As only values of -1 and 8 are used, it is possible to either subtract the value of the pixel, or add 8 times its value, respectively. This is shown in occam in figure 8.16.

```
SEQ y = 0 FOR no.of.rows
  SEQ x = 0 FOR no.of.cols
    B(x,y) := (A(x,y) << 3) -
                [ A(x-1, y-1) + A(x,   y-1) +
                  A(x+1, y-1) + A(x-1, y)   +
                  A(x+1, y)   + A(x-1, y+1) +
                  A(x,   y+1) + A(x+1, y+1) ]
```

Where "<< n" denotes bit shift left *n* places.

Figure 8.16. Example of Laplacian Convolution.

In figure 8.16, all the surrounding pixel values are added together, then the total subtracted from 8 times the value of the centre pixel value. The result is the transformed output image pixel value. This algorithm is dramatically faster than the general convolution, due to the simpler additions instead of multiplications and additions, as shown in table 8.5. Note the use of the binary shifts as a more efficient method of multiplying by 8 than "* 8".

Test input image	General Convolution	Laplacian Convolution
Wedge	713	459

Note: Image size 128 by 128. Timings in mS

Table 8.5. Convolution Timings.

Referring to figure 8.16, there would usually be some form of resultant pixel value range checking employed, to ensure that only valid pixel values were generated. Pixel values must be clipped to within the range *black* to *white*, so that any negative values would be set to black, and values above white set to white.

Image Edges. This Convolution algorithm does not take into account the edges of the image. Clearly at an edge of the image, neighbouring pixel values are not present. An attempt to access these would result in the boundary of the image array being violated, and an error condition being set up. A convenient solution is to assume that at edge points in the image, the neighbouring pixel values are the same as the edge pixel values. This has the advantage of not generating false object boundary edges when the edge of the image cuts through an object. In the example shown in figure 8.16, this could be taken into account by inserting extra occam code to deal with the first row, the starting and ending pixels on the other (no.of.rows - 2) rows, and the final row, as shown in figure 8.17.

```
SEQ
  SEQ x = 0 FOR no.of.cols
    ...   perform convolution on top row pixels A(x,0)
  SEQ y = 1 FOR no.of.rows - 2
    SEQ
      ...   left hand side pixel A(0,y)
      SEQ x = 1 FOR no.of.cols - 2
        ...   general centre pixel convolution, A(x,y)
      ...   right hand side pixel, A(no.of.cols-1,y)
  SEQ x = 0 FOR no.of.cols
   ...   perform convolution on bottom row pixels,
         A(x,no.of.rows-1)
```

Figure 8.17. Dealing with Edges in Convolutions.

Referring to figure 8.17, the convolution operation is performed initially on the first row in the image. On each

of the other rows, the left and right side pixels are treated separately, and the other pixels are transformed using a SEQ loop. Finally, the pixels on the bottom row are transformed.

Parallel Implementation of Laplacian Convolution. This operation was implemented using the SUPPLY and DEMAND processes on the multi-transputer network. The SUPPLY process divided the image into image sections, of size *N* by *M*, and supplied the neighbouring pixel values for each section automatically. In the parallel implementation, therefore, each DEMAND processed an image section comprising of *N + 2* rows by *M + 2* columns, operated on N by M pixels, and produced N by M transformed pixels. No special case conditions thus existed at image section boundaries, due to the neighbouring pixels being present.

Parallel Laplacian Results. Results obtained using three data section sizes are shown in tables 8.6, 8.7, and 8.8, and plotted on a graph in figure 8.18. The image size was 128 by 128.

Number of Transputers	Execution Timing/mS	Performance Speed Up	Efficiency
1	637	1	100%
4	176	3.6	90.5%
10	85	7.5	75%
20	73	8.7	43.5%
32	65	9.8	30.6%

Table 8.6. Laplacian Performance, using 16 by 32 Data Section

Number of Transputers	Execution Timing/mS	Performance Speed Up	Efficiency
1	825	1	100%
4	217	3.8	95%
10	93	8.9	89%
20	63	13.1	65.5%
32	52	15.9	49.7%

Table 8.7. Laplacian Performance, using 16 by 16 Data Section.

Number of Transputers	Execution Timing/mS	Performance Speed Up	Efficiency
1	874	1	100%
4	236	3.7	92.5%
10	105	8.3	83%
20	67	13	65%
32	69	12.7	39.7%

Table 8.8. Laplacian Performance, using 8 by 16 Data Section.

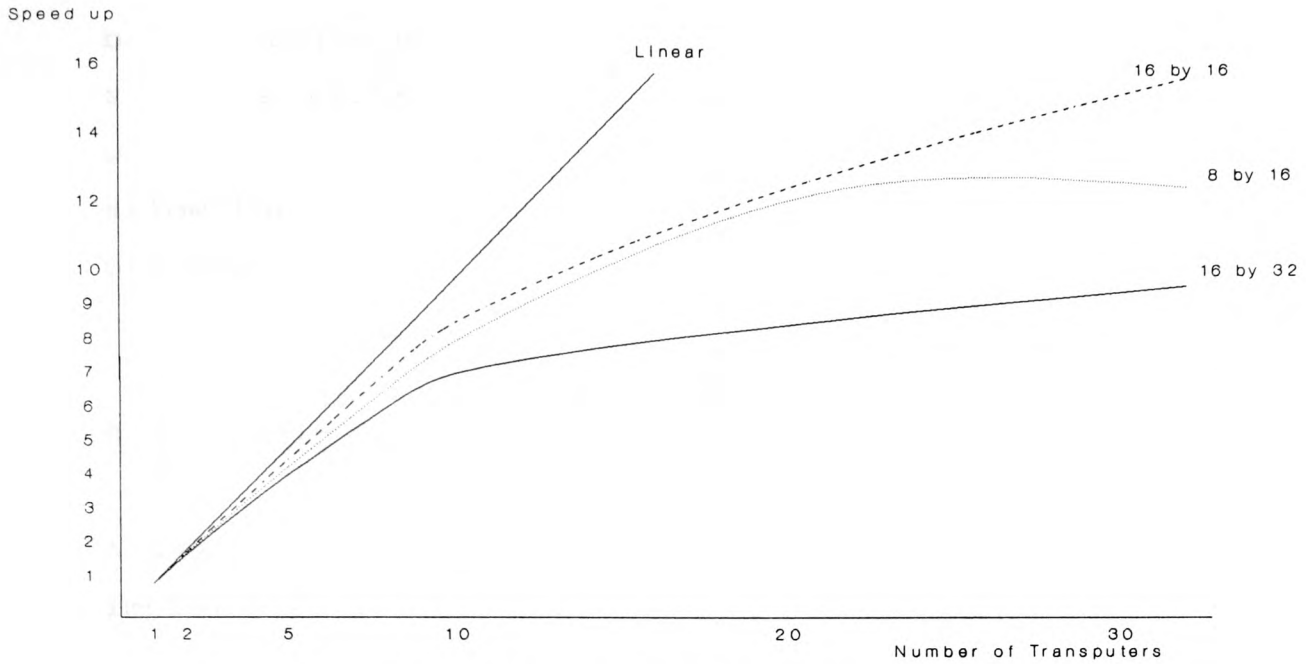


Figure 8.18. Laplacian Speed Up Against Number Of Transputers.

Discussion of Results. Referring to Figure 8.18, the speed ups obtained for the Laplacian transformation against different numbers of transputers can be seen for different image section sizes. The performance gains are approximately 86% of linear up to 4 transputer, and 80% of linear up to 13 transputers. With more than 13 processors, however, the performance gain deteriorates to 25% of linear, due to the extra level of nodes in the tree network, and the amount of data being used. The two turning points on the graphs, at 4 and 13 transputers, are due to the extra layers of processors being used in the tree network.

The performance of the 8 by 16 data section can be seen to be almost the same as for the 16 by 16 section, using up to about 20 transputers. Thereafter the performance increases obtained are degraded, due to the extra overhead pixels

associated with this data section size. A 8 by 16 section has 40.6% overhead neighbouring pixel values, while the 16 by 16 section has 26.6% (Table 14.1). The 16 by 32 section size has 19.5% overhead, but only produces 32 sections, which does not allow a sufficiently low granularity to allow the algorithm to be efficiently divided between the processors.

8.11 Dyadic Operations.

A useful dyadic operation, that of Subtract Images, may be implemented in occam as shown in figure 8.19.

```
SEQ y = 0 FOR no.of.rows
  SEQ x = 0 FOR no.of.cols
    C(x,y) := A(x,y) - B(x,y)
```

Figure 8.19. Dyadic Operation in occam.

As with the convolution example, some amplitude checking of the resultant pixel values must be performed, to ensure that only values between *black* and *white* are generated.

8.12 Local Operator Transformations.

Local operator transformations, where each pixel in the input image is replaced by a function of itself and its neighbouring pixels, can be realised using a straightforward technique, or by the use of a look up table. As with the Thresholding example, the look up table algorithm is the most effective, and could be implemented using hardware.

One useful local operator is that of Dilation, (Expand White Areas). Each pixel in the input image that is white, or has a one of its 8 neighbouring pixels white, is set to white.

Labelling the neighbouring pixels as:

a	b	c
d	x	e
f	g	h

The straightforward occam realisation of this algorithm is shown in figure 8.20.

```
SEQ y = 0 FOR no.of.rows
  SEQ x = 0 FOR no.of.columns
    IF
      A(x,y) = black
        B(x,y) := black
      (A(x,y)       = white) OR     -- centre pixel
       (A(x-1,y-1) = white) OR     -- pixel a
       (A(x,y-1)    = white) OR     -- pixel b
       (A(x+1,y-1) = white) OR     -- pixel c
       (A(x-1,y)    = white) OR     -- pixel d
       (A(x+1,y)    = white) OR     -- pixel e
       (A(x-1,y+1) = white) OR     -- pixel f
       (A(x,y+1)    = white) OR     -- pixel g
       (A(x+1,y+1) = white)        -- pixel h
        B(x,y) := white
      TRUE
        B(x,y) := A(x,y)
```

Figure 8.20. Straightforward Method of Local Operator.

It is interesting to note that the initial comparison of the centre pixel was modified to use the value of black, and was found to be more efficient than a comparison to white. This is because the constant black (0) is loaded into the transputer in one load of four bits, whereas white (255) requires two loads of four bits each, to load the 8-bit constant. The centre SEQ loops have again been

unrolled, for a faster algorithm.

The practical uses of Erosion and Dilation. This local operator, Dilation, is useful in eliminating small black areas, such as holes or imperfections in an image.

This algorithm could also be modified to yield an Erosion, (Shrink White areas) local operator (Expand Black areas), where a pixel is set to black if it or any of its neighbouring pixels are black.

It is interesting to note the effect of combining these two operators [9]. A Dilation operator followed by an Erosion operator eliminates small black areas, such as holes, cracks, and defects in the image. Such areas are effectively "filled in", but the rest of the object is unchanged. Areas of one or two pixel width will be filled in, other larger areas remaining unchanged. Performing an Erosion followed by a Dilation erases small clusters of pixels, thin lines, and noise. As with the other sequence, only clusters and lines of one or two pixel widths will be affected, the rest of the object remaining the same.

If more than one of each operator is used in the two sequences, then areas of more than two pixel widths can be eliminated. Combining two Dilation operations followed by two Erosion operations results in black holes and cracks of up to four pixel widths being eliminated, and consequently, inverting this sequence enables clusters, lines and noise of up to four pixels widths to be erased.

If the results of applying one of these operators are compared with the original, using a dyadic operator such as Exclusive-Or, Maximum, or Subtract, then edges will be

produced. The edges will either be part of white objects, if Erosion was used, or around white objects, if Dilation was used. If the results of sequences of such operators are compared with the original image, the areas previously eliminated can be studied, allowing defects and cracks to be examined on their own.

Local Operators using a Look Up Table. A local operator, such as the one featured above, may be realised using a look up table. Each of the 8 neighbouring pixels form part of an index into a previously calculated table, selecting the correct output pixel value.

Labelling the neighbouring pixels as:

a	b	c
d	x	e
f	g	h

and assigning bit masks as:

```
VAL a.bit IS #01 :
VAL b.bit IS #02 :
VAL c.bit IS #04 :
VAL d.bit IS #08 :
VAL e.bit IS #10 :
VAL f.bit IS #20 :
VAL g.bit IS #40 :
VAL h.bit IS #80 :
```

and assuming that only binary images are being considered at this stage, then this may be realised in occam as in figure 8.21.

```
SEQ y = 0 FOR no.of.rows
  SEQ x = 0 FOR no.of.columns
    IF
      A(x,y) = black
        B(x,y) := black
      TRUE
        SEQ
          index := A(x-1,y-1) /\ a.bit) \/ -- pixel a
           (A(x,y-1)   /\ b.bit) \/        -- pixel b
           (A(x+1,y-1) /\ c.bit) \/        -- pixel c
           (A(x-1,y)   /\ d.bit) \/        -- pixel d
           (A(x+1,y)   /\ e.bit) \/        -- pixel e
           (A(x-1,y+1) /\ f.bit) \/        -- pixel f
           (A(x,y+1)   /\ g.bit) \/        -- pixel g
           (A(x+1,y+1) /\ h.bit)           -- pixel h
            B(x,y) := look.up.table [ index ]
```

Where "/\" represents a bitwise logical AND operation, and "\/" represents a bitwise logical OR operation.

Figure 8.21. Local Operator using a Look Up Table.

This realisation relies upon the fact that with a binary image, black is 0, and white is 255, or #FF, (assuming that 8-bits per pixel representation is being used). An 8-bit index is built up, using one of each of the 8 neighbouring pixels to form each bit. Pixel *a* forms the least significant bit, and pixel *h* forms the most significant bit. The index is used to address the output pixel from a 256 element table, which must be calculated in advance.

Although the look up table algorithm may seem to be more complicated than the former technique, and hence would appear to take longer to execute, it was found that a significant speed improvement was obtained, as shown by table 8.9.

Test Figure	First Technique	Look Up Table	Speed Increase
zero image	547	526	1.04
All white	268	159	1.69
Binary Mug	476	437	1.09
Chess Board	394	343	1.15

Note. Timings for 128 by 128 image on one processor. All timings are in mS.

Table 8.9. Dilation Implementation Results.

The results in table 8.9 show that depending upon the test image data, the speed of execution of the Dilate local operator was increased by 1.04 times to 1.69 times faster.

Look up tables were employed for the computation of several other local operators, including Erosion, Binary Edge, Point Remove, and Count White Neighbours.

Parallel Implementation of Dilation Results. The results obtained for the parallel implementation of Dilation are shown in tables 8.10, 8.11 and 8.12, using three data section sizes and an image of size 128 by 128.

Number of Transputers	Execution Timing/mS	Performance Speed Up	Efficiency
1	501	1	100%
4	147	3.4	85%
10	69	7.3	73%
20	63	8.0	40%
32	64	7.8	24%

Table 8.10. Dilation Performance, using 16 by 32 Data Section.

Number of Transputers	Execution Timing/mS	Performance Speed Up	Efficiency
1	632	1	100%
4	171	3.7	92.5%
10	79	8	80%
20	58	10.9	54.5%
32	50	12.6	39.4%

Table 8.11. Dilation Performance, using 16 by 16 Data Section.

Number of Transputers	Execution Timing/mS	Performance Speed Up	Efficiency
1	687	1	100%
4	191	3.6	90%
10	92	7.5	75%
20	67	10.3	51.5%
32	66	10.4	32.5%

Table 8.12. Dilation Performance, using 8 by 16 Data Section.

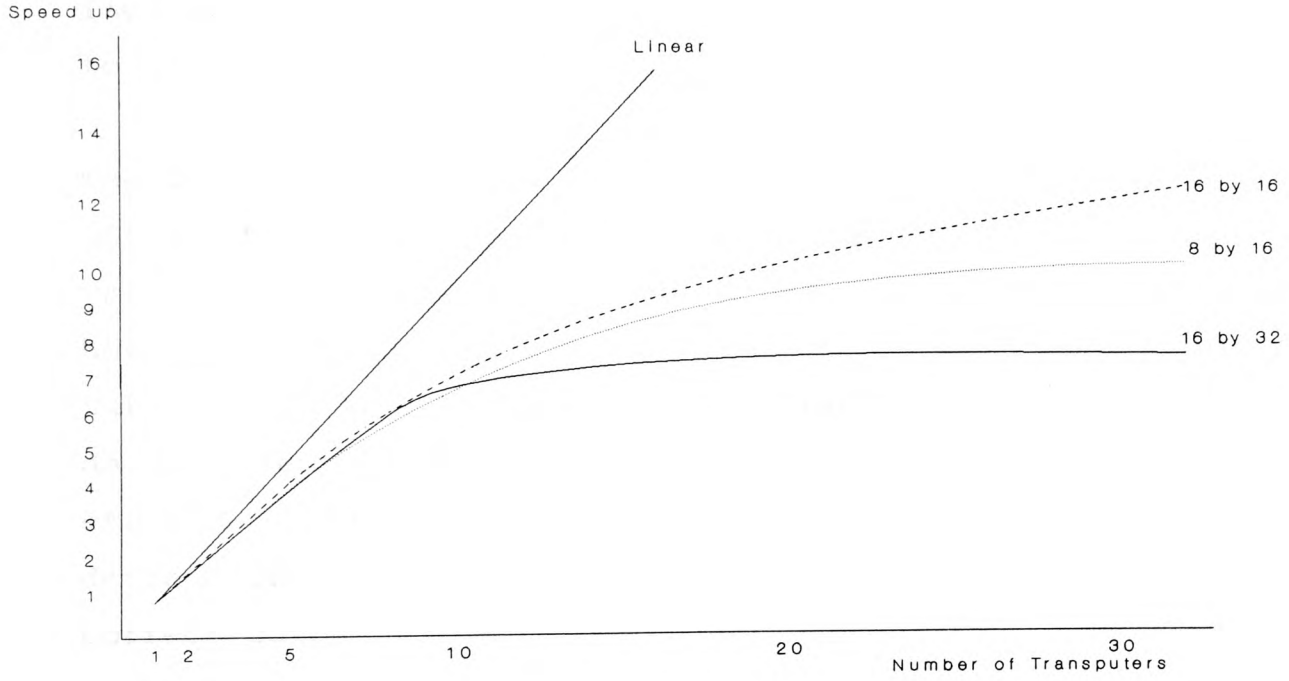


Figure 8.22. Dilation Speed Up Against Number Of Transputers.

Discussion of Results. Referring to figure 8.22, the speed ups obtained for the Dilation algorithm can be seen, using different image section sizes. The speed up using a 16 by 16 section was found to be approximately 86% of linear, using up to four transputers. The other section sizes used

were a little below this. This figure drops to 23% using 18 to 20 processors. This performance degradation is due to the amount of data being used with the operation (including neighbouring pixel values), as with the Laplacian Convolution operation.

8.13 Binary Edge

A binary edge routine was implemented using a modified form of the Laplacian Convolution algorithm described above. As the input image was assumed to be binary, a look up table implementation was possible, and this algorithm was found to be considerably faster than the Laplacian algorithm.

The look up table data had to be defined, for each of the 256 different combinations of the 8 neighbouring pixel values. Some examples are given here, in order to describe how the data was determined. With the previous look up table algorithms, the look up table data was readily defined, as Dilate has a straightforward data table, for example. The Binary Edge table data, however, must be defined in each of the 256 combinations, in order that the correct output value is obtained.

Consider the neighbouring pixel values in the combinations shown in figure 8.23. A binary bit masking used in the look up table algorithm produces single bit values used to generate the 8-bit binary index. The centre value is assumed to be 1 in the examples, (white), as otherwise the neighbouring values do not need to be examined, and the output pixel can be set to 0, (black), regardless of their values.

```
(a)   0 0 0          (b)   1 1 1
      0   0  -> 0          1   1  -> 0
      0 0 0                1 1 1

(c)   1 1 0          (d)   1 1 1
      1   0  -> 1          1   0  -> 1
      1 1 0                1 0 0

(e)   1 1 0          (f)   1 1 1
      1   0  -> 1          0   0  -> 0
      0 0 0                0 0 0

(g)   0 0 0
      1   1  -> 1
      0 0 0
```

Note. Black is represented as 0, and white as 1.

Figure 8.23. Examples of Binary Edge Look Up Table Data. (a) and (b) produce 0. (c), (d), (e), (f) and (g) all have three other rotations, each of which produce the respective output shown.

Referring to figure 8.23, (a) and (b) represent all black and all white combinations respectively, the binary edge of these being 0, black. Examples (c) and (d) represent vertical and diagonal edges, which should produce 1, white, when passed through the binary edge operation. Some less straightforward examples are shown by (e), (f) and (g). It may be noted that more examples can be found by rotating most of those shown by successive increments of 90 degrees.

The results of using this look up table version of the Binary Edge operation on one transputer are presented in table 8.13.

Test figure	Modified Laplacian Version	Look Up Table Version
Zero image	483	182
All white	483	684
Binary mug	482	303
Chessboard	481	428

Note: Timings in mS.

Table 8.13. Results of using a Look Up Table for Binary Edge.

Table 8.13 clearly shows the advantages of using a look up table for generating a Binary Edge of an object. In the worst case, using an all white test image, the execution speed increased, to 0.71 of the original value, but in all other test images used, the execution speed was increased from 1.12 to 2.65 times faster. The most realistic test image used, that of a binary mug, the speed increased to 1.6 times faster.

Parallel Implementation of Binary Edge Results. The results obtained for the parallel implementation of Binary Edge are shown in tables 8.14, 8.15 and 8.16, using three data section sizes, and an image of size 128 by 128.

Number of Transputers	Execution Timing/mS	Performance Speed Up	Efficiency
1	314	1	100%
4	93	3.4	85%
10	57	5.5	55%
20	64	4.9	24.5%
32	64	4.9	15.3%

Table 8.14. Binary Edge Performance, using 16 by 32 Data Section.

Number of Transputers	Execution Timing/mS	Performance Speed Up	Efficiency
1	419	1	100%
4	113	3.7	92.5%
10	62	6.8	68%
20	48	8.7	43.5%
32	48	8.7	27.2%

Table 8.15. Binary Edge Performance, using 16 by 16 Data Section.

Number of Transputers	Execution Timing/mS	Performance Speed Up	Efficiency
1	489	1	100%
4	140	3.5	87.5%
10	76	6.4	64%
20	67	7.3	36.5%
32	68	7.2	22.5%

Table 8.16. Binary Edge Performance, using 8 by 16 Data Section.

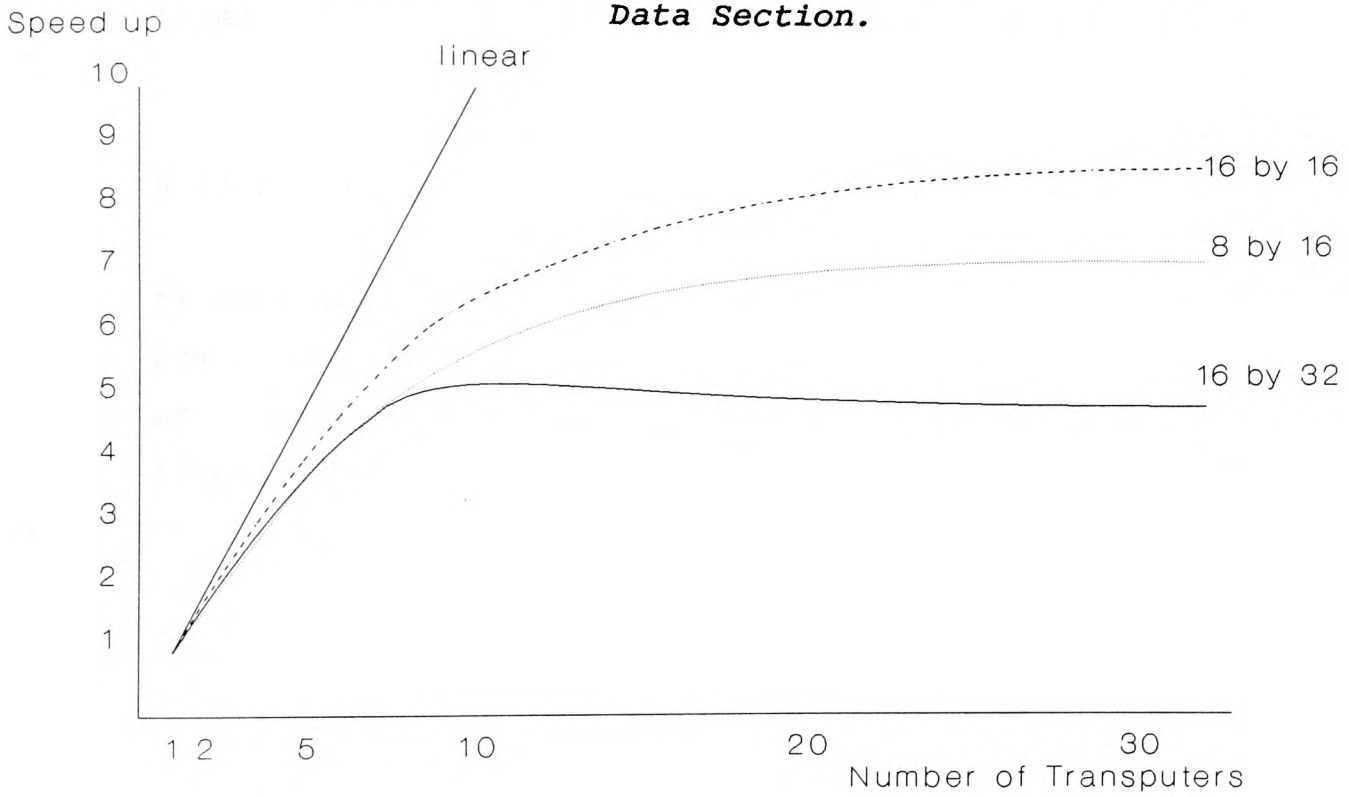


Figure 8.24. Binary Edge Speed Up Against Number Of Transputers.

Discussion of Results. The results are plotted in graphical form in figure 8.24. The Binary Edge operation was performed in 48 mS using a 16 by 16 section size, and either 20 or 32 transputers. The performance was found to

be 90% of linear using up to four transputers, and a section size of 16 by 16. This figure drops to approximately 22% using 14 to 16 processors, and actually evens out at 0 using 32 transputers. The performance increases for the other two section sizes were slightly below the figures for the 16 by 16 size.

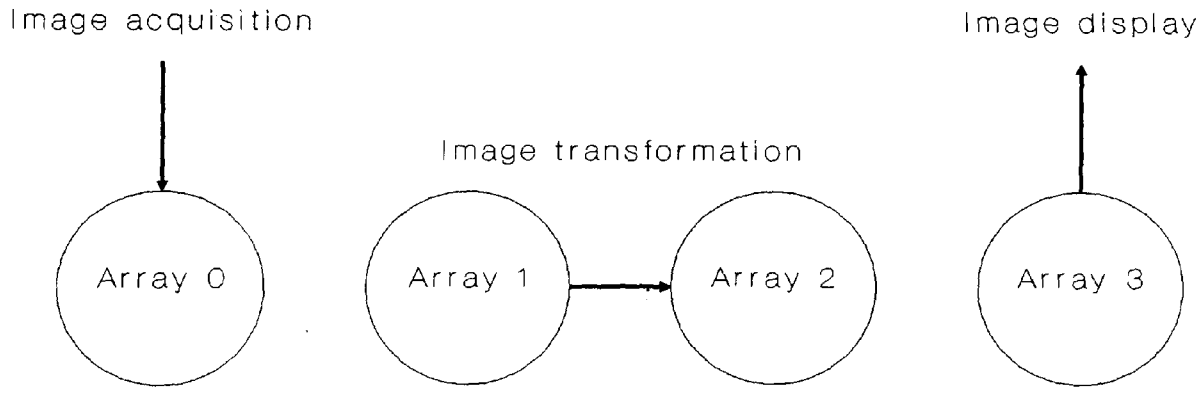
As with the Dilation and Laplacian results, the performance degradation using more than 4 and more than 13 transputers is due to the amount of data being used, the fairly low computational requirements of this operation, and the extra levels of nodes being used in the tree architecture.

8.14 Continuous Operations

It was found that image data capture and display could be performed in parallel with image transformation, when appropriate sequences of operations were implemented (Figure 8.25).

```
WHILE running

  PAR
    capture image 1
    display image 2
    process image 3 to image 4

  PAR
    capture image 2
    display image 4
    process image 1 to image 3

  PAR
    capture image 4
    display image 3
    process image 2 to image 1

  PAR
    capture image 3
    display image 1
    process image 4 to image 2
```

(a) Example Occam Structure.



(b) Array Usage of Overlapped Data Movements.

Figure 8.25. Image Data Capture, Display, and Transformation in Parallel.

In this way, images are continually captured from the camera, processed to form a new image, and displayed. These three basic operations can be performed in parallel, using four different images, so that image capture and

display times are effectively hidden, and the only time that is noticed is that of the image processing transformation itself.

Using this technique, images could be continually Edge Detected and Thresholded, say, and the results displayed. This could be useful in dealing with Radar image data, or preliminary robotic vision.

8.15 Streamlining

The Problem. When a sequence of operations was executed, in the form of a macro (described in Chapter 6), it was found that the implementation of the dynamic load balancing at the instant in time at which one operation completed, and the next one started was inefficient. This effect was less significant with data dependent operations (such as Feature Extraction) than it was with low level transformations.

The cause of the inefficiency was the distribution of the final data or image sections of operation *N*, say, prior to starting operation *N+1*. The worst case scenario would be for all but one DEMAND processes to have completed their work on operation *N*, while the last data or image section associated with operation *N* was being operated upon. This would result in all but one of the processors being kept idle until this computation had been performed (figure 8.26).

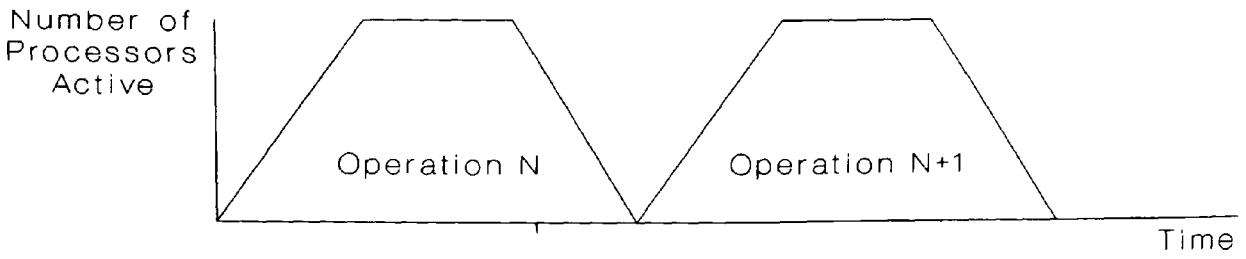


Figure 8.26. Inefficiency at Transitions Between Operations.

A Solution. In order to increase the efficiency of the network utilisation during this transition from one operation to the next, a technique was devised, termed *Streamlining.* This technique allowed operations to overlap, so that when data had been exhausted from operation *N*, while computation was still being performed, data that had been returned from this operation could be transmitted for the next operation, *N+1*, to be performed on it. This meant that while some processors were working on data associated with operation *N*, other processors were working on data produced from operation *N*, performing operation *N+1*. This procedure is illustrated in figure 8.27.

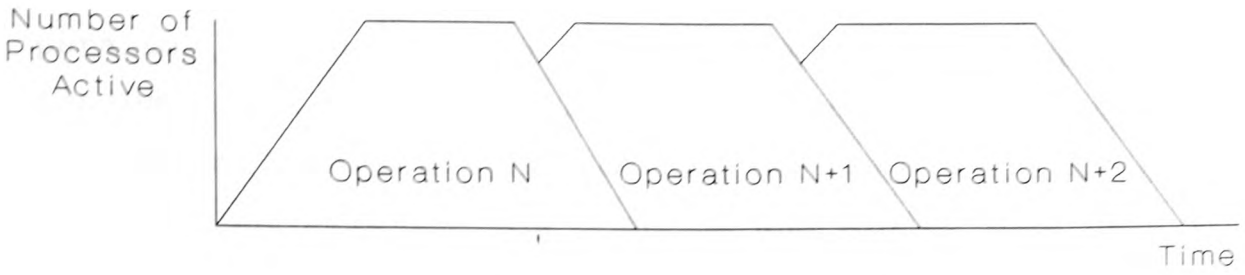


Figure 8.27. Overlap of Operations Using Streamlining Technique.

The amount of streamlining that was allowed depended upon several factors. The number of work packets, the amount of data dependence inherent in the operations, and the software complexity all determined the degree of Streamlining implemented and obtained.

Implementation of Streamlining. Streamlining was implemented with low level transformations, and the parallel Object Boundary and parallel Object Convex Hull generation.

With transformations, data sections that had been received by SUPPLY from operation *N*, could be distributed for operation *N+1*. There was no dependence on other sections, as was the case with parallel Boundary and parallel Convex Hull generation.

8.16 Conclusions

A wide variety of algorithms have been devised and realised in occam. Most of these were adapted to execute in parallel, on the ternary tree network of transputers. The implementation of Thresholding, Contrast Enhancement, Histogram Equalisation, General Convolution, Laplacian Convolution, Dilation and Erosion, and Binary Edge have all been described and discussed. Improvements in the algorithms have been featured, and their results presented.

Low level image processing transformations were implemented using the SUPPLY and DEMAND software architecture. Images were divided into rectangular sections of size 16 by 32, 16 by 16 or 8 by 16 for transmission to the DEMAND processes. The correct neighbouring pixel values were also transmitted, for those operations that required them, including Local Operator and Convolution transformations.

Results for these low level image processing algorithms have been presented, for a tree network involving 1 to 32 transputers. It can be seen that the highest speed ups were obtained using an image section of size 16 by 16. This yielded 64 data sections, and thus allowed a good degree of workload distribution.

The results obtained indicate a near linear speed up using four transputers in two levels of the tree. With up to four transputers, the speed up for the Negate operation was 70%, Laplacian 86%, Dilation 86%, and Binary Edge was 90%.

As more processors are added to the third layer, however, the speed ups results become far from linear. The figure for Laplacian drops to 80% using 13 transputers, and to 25%

with 25 processors. The speed up for Dilation drops to 23% using 20 processors, and that for Binary Edge drops to 22% with 16 processors.

The speed up obtained for these transformations was near linear using up to 4 transputers, but was found to be non-linear due to the large amount of data and low computational requirements inherent with these operations. The data must be passed down the tree for transformation, then transformed data passed back up the tree again. As there is a large amount of data being communicated around the system, the overall efficiency drops. This is due to two basic factors:

(a) The time taken to communicate an image section up or down the tree is significant compared to the time required to perform the computation. Using a 16 by 32 image section, the time to transmit the data over a T414 link at 10 Mbps is 1.28 mS. This has been reduced by the use of T800 devices, and using a 20 Mbps link speed, reducing this figure to approximately 310 μS, or about a quarter. The time required to pass a data section down the tree, however, remains significant.

The use of transputer devices having more links has been proposed in Chapter 1. This would enable the software used in this research, with some modifications, to be used with an higher order tree, depending upon the number of links. This would dramatically improve the results for the low level image processing transformations, as the tree would become smaller in depth for a given number of processors. It is predicted that the speed up obtained for four transputers would continue up to the full second level of the tree.

(b) While a DEMAND process is computing, it will be interrupted in order to receive then transmit an image section. The more data that is relayed in this fashion, the greater will be the impact upon the processor.

The tree is proposed as a suitable vehicle for the execution of higher level, data dependent algorithms, where a significant amount of non determinate computation must be performed, and the amount of data being used is kept to a minimum.

CHAPTER 9

IMPLEMENTATION OF BOUNDARY CHAIN CODING

9.1 Introduction

Several Feature Extraction algorithms were implemented, involving the distribution of $(N + 2)$ by $(M + 2)$ image sections, including the neighbouring pixel values, to the DEMAND processes. The Feature Extraction operations included Boundary Chain Code, Convex Hull, Areas and Perimeters, Convex Hull Deficiencies, Shape Factors, and Convex Hull Deficiency area to object area ratios.

These operations produced not two dimensional transformed image data, but either sequences of numbers comprising chain codes representing object boundaries or Convex Hull boundary constructions, or numbers representing area or perimeter values.

This Chapter presents the implementation of Boundary chain coding on the multi-transputer configuration. This is useful, as not only does it greatly reduce the amount of data required to describe an object, but the technique renders the object description into a data format suitable for extracting features, necessary in any object inspection or recognition task.

9.2 Image Work Distribution

Medium level feature extraction algorithms are highly data dependent. Some parts of an image may contain little or none of an object, in which case there would be little or no computational work associated with them (figure 9.1). Sections of an image that include objects, however, will involve a great deal of computation, to identify and extract features of interest.

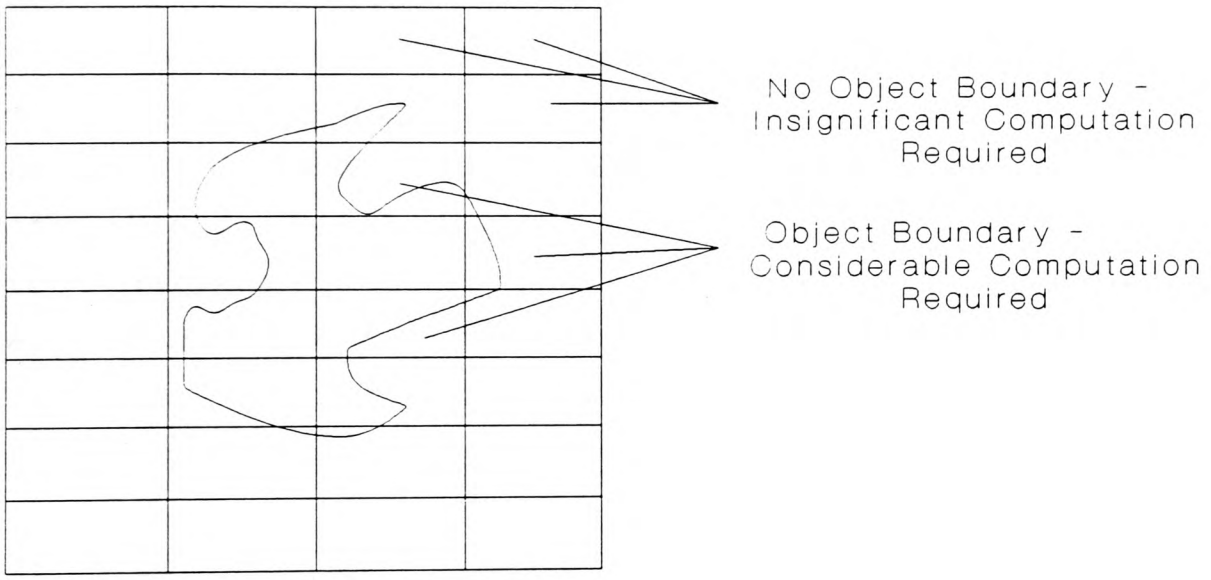


Figure 9.1. Typical Uneven Workload Distribution in an Image.

Different objects will generally have different shapes, sizes, orientations and locations within images. There will usually be significant differences, therefore, in the geometric distribution of the computational load from one image to another. This is entirely indeterminate, as the data in images is only available at run time.
These factors make static load balancing extremely

inefficient. If an array is used to execute these operations, where each processor deals with the image section corresponding to its position in the array, then clearly the workload will be unevenly distributed. Just as some sections of the image will have little or no work associated with them, so those respective processor nodes will have little or no work to do. Referring to figure 9.1, it can be seen that, for this simple example, out of the 32 data sections, 17 sections, or 53% are outside the object and have no boundary edges present. Of the remaining sections, 14 or 44% have boundary edges present, and one section (3%) contains some of the object, but does not actually have any boundary edges present. It is clear, therefore, that a geometric mapping of this object onto an array of processors could, at the very best, achieve an efficiency of 44%. Within the sections that do have boundary edges present, however, the amount of computation associated with them can be seen to be unequal, resulting in the theoretical maximum efficiency dropping significantly from 44%.

It is proposed that dynamic load balancing techniques are essential in these circumstances. These techniques allow the computational work contained in the image to be divided and spread between the available processor elements. As image sections are processed, the results are passed back to the root SUPPLY process, and another image section is passed to the appropriate node.

These feature extraction operations were implemented on the ternary tree configuration using the SUPPLY and DEMAND software architecture. The dynamic load balancing achieved allowed a more even allocation of work to all the processor nodes in the system.

9.3 Chain Coded Boundaries - Initial Algorithm

The chain code of an object's boundary was generated in parallel throughout the multi-transputer system. Each DEMAND process was given rectangular sections of the grey level image by SUPPLY, with the neighbouring pixel values present, as with the low level algorithms described in Chapter 8. The image section was converted into a binary representation, using a Thresholding operation, and subsequently coded using the standard chain coding representation [142] (figure 9.2).

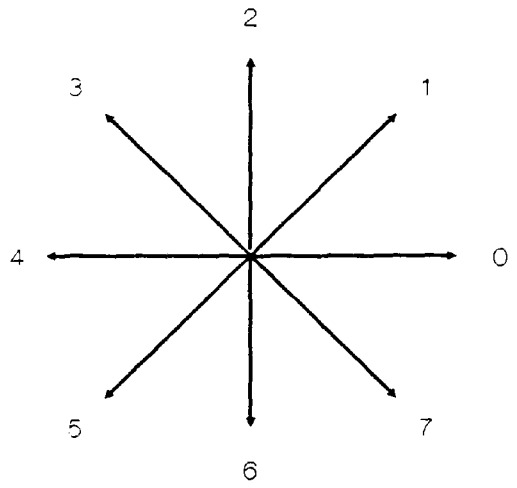


Figure 9.2. Chain Coding Direction Codes.

The initial algorithm for the chain coding used a neighbourhood scanning technique, operating on binary edges in the image section produced using a local Automatic Threshold transformation, and a local Binary Edge operator. When a white pixel element was found, using a raster scan technique, its eight neighbouring pixels were scanned, in order to track the edge. If one of these was also white, then the appropriate direction code was inserted into the chain, and that pixel's neighbours scanned. If no other white pixel was found, then the edge was deemed to have ended, and another start of an edge had to be found.

This algorithm had the disadvantages that edges were not necessarily coded in their correct directions. In the worst case, an edge in a section could produce two chains, necessitating a local chain joining stage (figure 9.3).

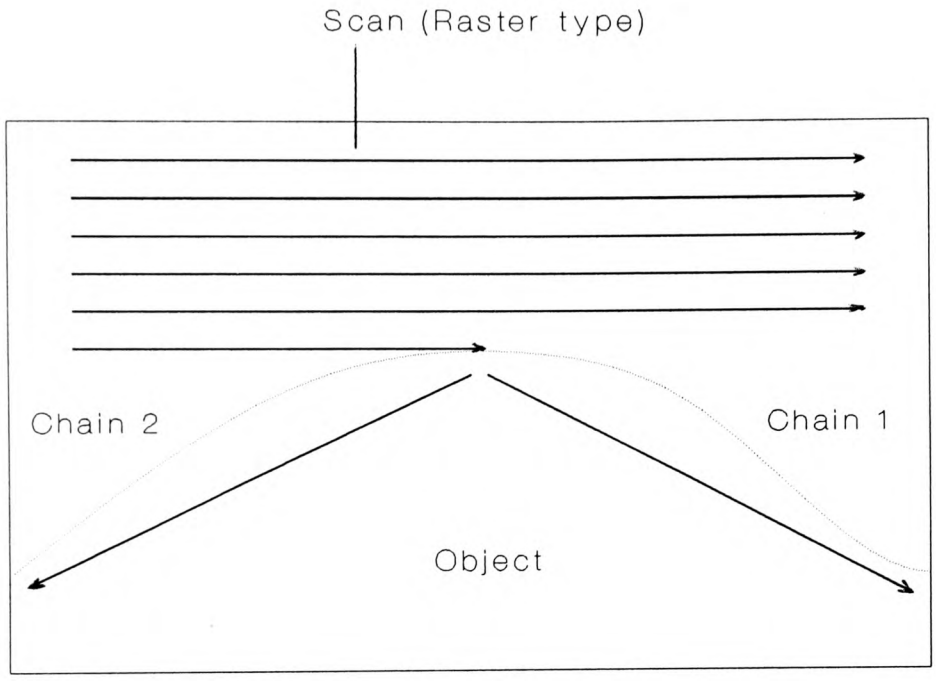


Figure 9.3. Initial Chain Coding Algorithm - Worst Case.

To test two chains for joining, it was necessary to adopt the first chain's (x.start, y.start) and (x.end, y.end) coordinates, and compare these with the start and end points of the second chain. This was achieved using nested replicated IF constructs, as illustrated in figure 9.4.

```
IF
  IF delta.x = -1 FOR 3     -- test start.1 joins to start.2
    IF delta.y = -1 FOR 3
      ((x.start.1 + delta.x) = x.start.2) AND
       ((y.start.1 + delta.y) = y.start.2)
        ...   perform joining

  IF delta.x = -1 FOR 3     -- test start.1 joins to end.2
    IF delta.y = -1 FOR 3
      ((x.start.1 + delta.x) = x.end.2) AND
       ((y.start.1 + delta.y) = y.end.2)
        ...   perform joining

  IF delta.x = -1 FOR 3     -- test end.1 joins to start.2
    IF delta.y = -1 FOR 3
      ((x.end.1 + delta.x) = x.start.2) AND
       ((y.end.1 + delta.y) = y.start.2)
        ...   perform joining

  IF delta.x = -1 FOR 3     -- test end.1 joins to end.2
    IF delta.y = -1 FOR 3
      ((x.end.1 + delta.x) = x.end.2) AND
       ((y.end.1 + delta.y) = y.end.2)
        ...   perform joining

  TRUE
    ...   these two chains will not join
```

Figure 9.4. Nested IF Constructs Testing For Joining Chains.

This joining was complicated by the fact that two chains could be joined in one of four possible ways, start to start, end to end, start to end, and end to start. Furthermore, if chains were joined start to start or end to end, then one of the chains had to be reversed, as its direction would be opposite to the other. This chain reversal was achieved using an addition of 4 modulo 8:

reversed chain code := (old chain code + 4) \ 8
(where \ denotes modulo)

In practice, this function was provided as a simple look up table.

Having the chains coded in either direction proved to be a significant problem when dealing with longer chains, as typically quite lengthy chains had to be reversed.

Whenever two chains were joined, an extra direction code had to be produced, to link them together. This was computed using a look up table (figure 9.5).

```
--Start to Start
VAL [3][3]INT new.dir.ss IS [[se,e,ne], [s,8,n], [sw,w,nw]] :
new.chain.code := new.dir.ss [ delta.x + 1 ] [ delta.y + 1 ]

--Start to End
VAL [3][3]INT new.dir.se IS [[se,e,ne], [s,8,n], [sw,w,nw]] :
new.chain.code := new.dir.se [ delta.x + 1 ] [ delta.y + 1 ]

--End to Start
VAL [3][3]INT new.dir.es IS [[nw,w,sw], [n,8,s], [ne,e,se]] :
new.chain.code := new.dir.es [ delta.x + 1 ] [ delta.y + 1 ]

--End to End
VAL [3][3]INT new.dir.ee IS [[nw,w,sw], [n,8,s], [ne,e,se]] :
new.chain.code := new.dir.ee [ delta.x + 1 ] [ delta.y + 1 ]
```

Note: "n" represents *North* (Direction code 2), "s" represents *South* (Direction code 6) etc. "8" represents *not.a.code* (a non-valid direction code).

Figure 9.5. Computation of New Chain Code.

Referring to figures 9.4 and 9.5, the new chain code necessary to join two chains was found in the appropriate look up table using the *delta* offsets that were used to determine that the two chains would join. The data for the look up tables was determined by consideration of the four possible joining cases.

9.4 Improved Algorithm Using a Look Up Table

The chain coding algorithm was improved, by utilising a look up table technique, similar to that described in Chapter 8 for the Binary Edge operation. This new algorithm had the advantages of a fast execution, always produced coded edges in their correct direction, and was more structured and therefore maintainable.

The data had to be specified for the look up table, to determine which (if any) direction code would be produced for each of the 256 different neighbouring pixel combinations. Some example neighbouring pixel combinations are illustrated in figure 9.6, together with their resultant directional chain code.

Referring to figure 9.6, eight combinations out of the 256 possible cases are illustrated. The centre pixel in each case must be "1", for the neighbours to be examined. As a binary image was operated on, single lines of pixels could be discarded. In the look up table, it was found that 44 neighbourhood combinations generated a valid direction code, with the remaining 212 generating *not.a.code* (a non-valid direction code).

0	1	1				1	1	0			
0		1	->	2		1		0	->	6	
0	1	1				1	1	0			
0	0	0				1	1	1			
1		1	->	0		1		1	->	4	
1	1	1				0	0	0			
0	0	1				1	1	1			
0		1	->	1		0		1	->	3	
1	1	1				0	0	1			
1	0	0				1	1	1			
1		0	->	7		1		0	->	5	
1	1	1				1	0	0			

Note. Black is represented as 0, and white as 1.

Figure 9.6. Example Data in Look Up Table For Chain Coding.

Once edges had been passed through the look up table, the image section then comprised of direction codes in place of the pixel values. These then had to be put into a sequence which would be the chain representing that edge.

The image section was scanned around the border initially, as most edges present would intersect the edge of the section. When a direction code was found, the start coordinates were noted, then the direction codes were tracked, copied, and set to *not.a.code* in the image section. The chain ended when no direction code was found in the appropriate position. The border scan was then continued, from the position where the last chain started, until no other direction codes were encountered. A raster

scan was then performed, to ensure that no small complete objects, such as holes, were missed. These chains generated were termed Partial Boundary (PB) Chains.

After generation of the local Partial Boundary Chains, a local Join Up routine was executed, which ensured that no chains were left not joined into a single chain that could be. Such small chains were individual direction codes found during the perimeter scan that pointed the wrong way, i.e. were at the start or end of a Partial Boundary Chain. Figure 9.7 illustrates a worst case scenario with this algorithm. As can be seen, only single element chains need to be joined to the main chains generated.

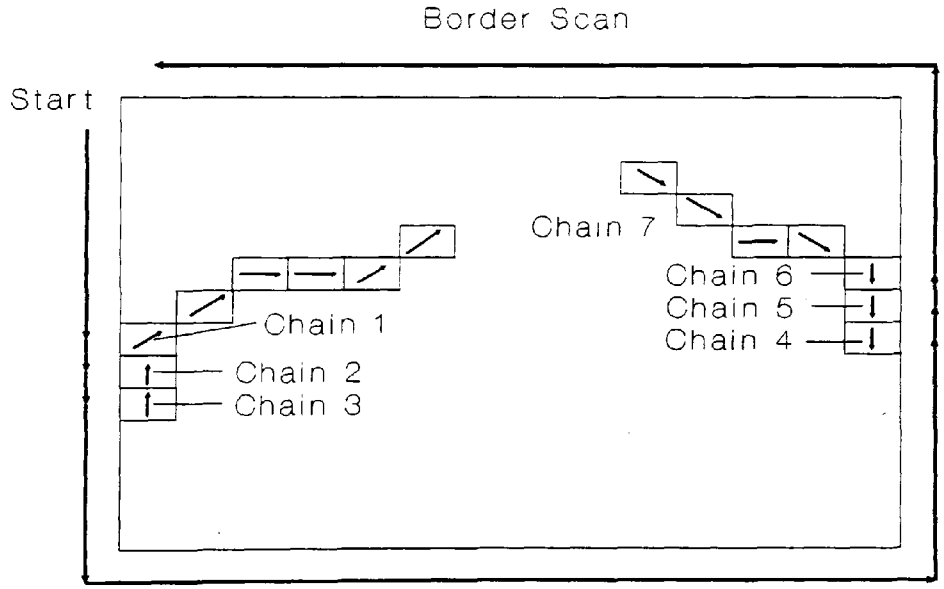


Figure 9.7. Improved Chain Coding Algorithm - Worst Case.

Objects were always coded clockwise and holes in objects coded anti-clockwise. Each chain had its X and Y start & end co-ordinates included, along with a chain identity (ID), generated from the identity of the respective image section.

9.5 Joining Up Partial Boundary Chains

Partial Boundary Chains, after being joined locally where possible, were passed to the root SUPPLY process. Chains produced using the look up table algorithm could only be linked to other chains in one of two ways, end to start, or start to end. Chains could not be considered for joining in the other two combinations, end to end or start to start, as with the initial algorithm, due to their inherent correct directionality.

It was possible to allow for non-continuous edges, due to quantisation errors or noise, by allowing chains to be joined with one or two direction codes missing. To allow for single missing direction codes, replicated IF constructs were used, as shown in figure 9.8.

Referring to figure 9.8, a neighbouring 5 by 5 window, centred on the start and end points of a chain, was tested, to determine whether another chain can be joined. Two extra direction codes must be produced, to link up the two chains, if they will join. Figure 9.9 illustrates this procedure. A 7 by 7 window search was also implemented, in order to allow chains to be joined with up to two links missing, therefore requiring three new direction codes to be generated, but in practice this was found unnecessary.

```
IF
  -- test start.1 to end.2, 1 code missing
  IF delta.x = -2 FOR 5
    IF delta.y = -2 FOR 5
      ((x.start.1 + delta.x) = x.end.2) AND
       ((y.start.1 + delta.y) = y.end.2)
        ...   perform joining

  -- test end.1 to start.2, 1 code missing
  IF delta.x = -2 FOR 5
    IF delta.y = -2 FOR 5
      ((x.end.1 + delta.x) = x.start.2) AND
       ((y.end.1 + delta.y) = y.start.2)
        ...   perform joining

  TRUE
    ...   chains will not join
```

Figure 9.8. Testing Chains For Joining With Links Missing.

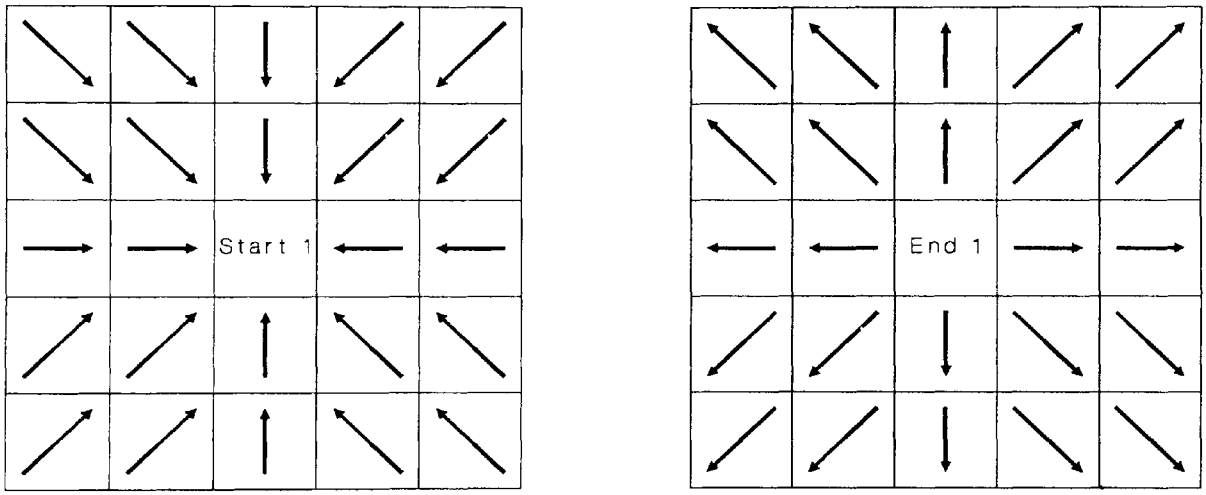


Figure 9.9. Scanning Within a 5 by 5 Neighbouring Window.

When joining two chains with a link missing, two new direction codes had to be generated, in order to link them. These were provided using three-dimensional look up tables, as shown in figure 9.10.

```
--start to end
VAL [2][5][5] INT new.dir.55se IS [[[se,se,e,ne,ne],
                        [se,se,e,ne,ne], [s,s,8,n,n],
                        [sw,sw,w,nw,nw], [sw,sw,w,nw,nw]],
                        [[se,e,e,e,ne], [s,8,8,8,n],
                        [s,8,8,8,n], [s,8,8,8,n],
                        [sw,w,w,w,nw]]] :
first.new.chain.code :=
  new.dir.55se [ 0 ] [ delta.x + 2 ] [ delta.y + 2 ]

second.new.chain.code :=
  new.dir.55se [ 1 ] [ delta.x + 2 ] [ delta.y + 2]

--end to start
VAL [2][5][5] INT new.dir.55es IS [[[nw,nw,w,sw,sw],
                        [nw,nw,w,sw,sw], [n,n,8,s,s],
                        [ne,ne,e,se,se], [ne,ne,e,se,se]],
                        [[nw,w,w,w,sw], [n,8,8,8,s],
                        [n,8,8,8,s], [n,8,8,8,s],
                        [ne,e,e,e,se]]] :
first.new.chain.code :=
  new.dir.55es [ 0 ] [ delta.x + 2 ] [ delta.y + 2 ]

second.new.chain.code :=
  new.dir.55es [ 1 ] [ delta.x + 2 ] [ delta.y + 2]
```

Note: "n" represents *North* (Direction code 2), "s" represents *South* (Direction code 6) etc. "8" represents *not.a.code* (a non-valid direction code).

Figure 9.10. Computation of Two New Chain Codes.

9.6 Distributed Joining Up of Partial Boundary Chains

All processing was now performed on chains, and not two dimensional image sections. When Partial Boundary chains were returned to the root SUPPLY process, they had to be joined to other chains produced from other image sections. This involved invoking a joining algorithm within the DEMAND processes, which attempted to join non-closed chains generated from *adjacent* image sections.

Chains were distributed by SUPPLY to the DEMAND processes for joining. The Partial Boundary chains produced by the

first stage were stored in a two dimensional array, indexed using the chain identity.

Adjacent Image Sections. Only Partial Boundary chains produced from horizontally and vertically adjacent image sections need be considered for joining. It would also be possible for a boundary to cross diagonally adjacent image sections at the exact corner point of contact, in which case diagonally adjacent sections could have been considered also. It would have been more efficient (but inconsistent) to distribute chains from the appropriate diagonally adjacent sections in this case. Very little additional overhead was introduced by disregarding this special case, however, as it was observed to be an extremely rare situation and was therefore not deemed worthy of special consideration.

Stage 2 consisted of sending out pairs of chains produced from these adjacent image sections in stage 1 to the DEMAND processes. Groups of 4 such sections were sent out together, and processed by a joining and PCH computation process within each DEMAND process. The identification of adjacent image sections is illustrated in figure 9.11.

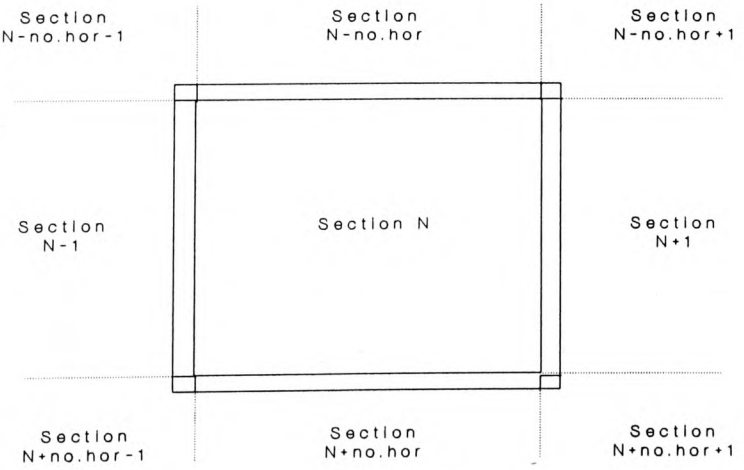


Figure 9.11. Identification of Adjacent Image Sections.

A formula was devised that yielded the four stage 1 identities from any stage 2 identity, and for the 4 stage 2 identities from a stage 3 identity. If *stage.2.id* is the identity of the second stage position then the four stage 1 chain identities *a,b,c* and *d* required are shown in figure 9.12.

```
a := ((stage.2.id / no.hor) * (no.hor * 2)) +
     (stage.2.id \ no.hor) * 2)
b := a + 1
c := a + no.hor
d := c + 1
```

Where *no.hor* is the number of stage.1 sections horizontally.

Figure 9.12. Formulae To Calculate Logical Image Sections.

To distribute chains produced from adjacent image sections, (or adjacent larger logical image sections), therefore, the appropriate identities were calculated, and the chains in that dimension of the array extracted.

The joining was achieved within each DEMAND process by firstly adopting the start and end coordinates of the first chain in the table, and comparing them with the start and end points of the other chains. When a chain was found that would join to the first, new start or end coordinates were computed, to allow the new chain produced to be compared with other chains in the table. A linked list was generated, which contained the order and joining information about chains joined.

When all the chains had been examined, the data comprising the chains that had been joined was extracted from the

table, in the appropriate order, forming a new chain. The new, longer chain was copied into an output table, with its identifier set to that of the larger *logical* image section containing the four initial image sections. Any remaining chains in the table then had to be compared, for possible joining. In this way, unnecessary chain data copying was avoided, as the data pertaining to a chain was only copied when it had either been joined with others, and no more chains would join onto the new, longer chain, or all the chains had been compared, and the chain would not join onto any of them.

The chains produced by this first stage of the distributed joining algorithm were returned to SUPPLY. The joining process was reiterated, using increasingly large logical image sections. Joined Partial Boundary chains that were closed, so that the end of the chain joined the start, were complete, and therefore were not needed in any further joining.

The boundary chain code generation was achieved using several stages. Using a 128 by 128 image, and 16 by 32 image sections, four stages were required. The local Partial Chain generation from image data, joining four such regions' chains, joining four groups of four regions' chains, and finally joining the two image halves' chains.

These stages were overlapped, so that while the final computation was being performed by DEMAND processes for stage *N*, for example, computation was also being performed, in parallel, by other DEMAND processes, for stage *N + 1*. The image sections and larger, logical groups of sections are illustrated in figure 9.13.

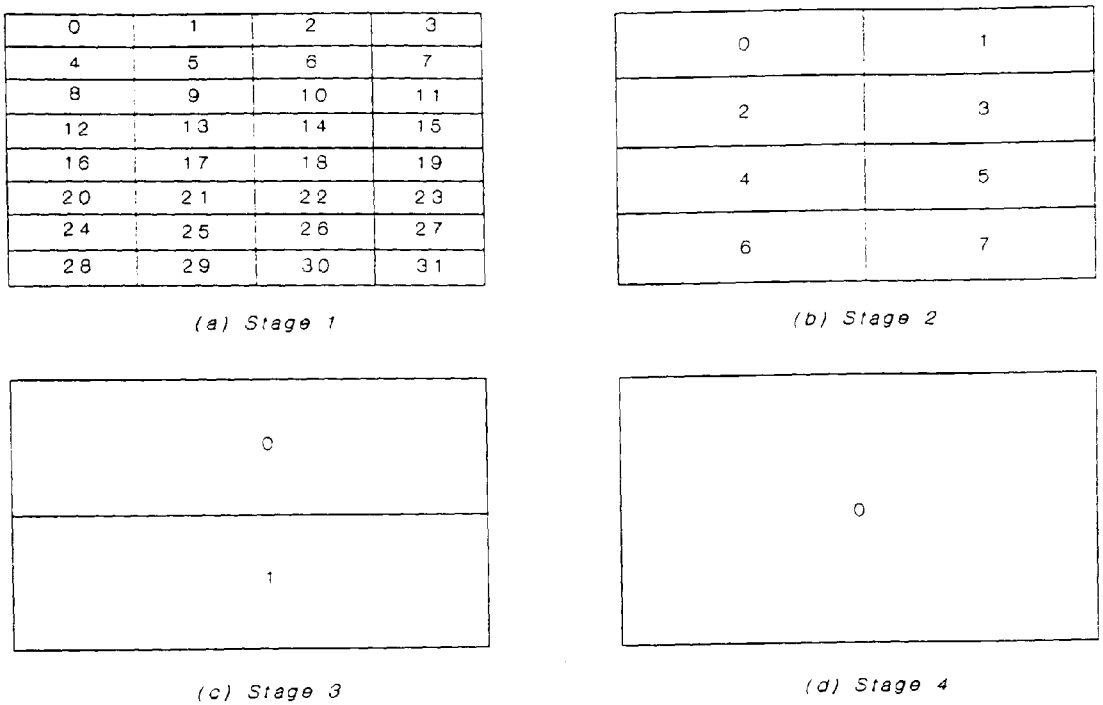


Figure 9.13. The Stages in the Distributed Boundary Chain Code Generation.

The boundary chain for an object was thus complete when a closed chain had been obtained. Non-closed or short boundary chains representing noise, unimportant background objects, or unwanted non-object related edges could be discarded.

9.7 Streamlining of Distributed Join Up Stages

The distributed join up stages were overlapped, using the Streamlining technique described in Chapter 8. This allowed DEMAND processes to work on stage 2, while other DEMAND processes were still working on stage 1. It was possible for DEMANDs to work on stage 3 while stage 2 was also being completed.

Score Chart. A "score chart" was utilised, to ensure that

the appropriate Partial Boundary chains had been received from stage *N*, say, that were necessary for stage *N+1*. As an example, consider in figure 9.13, the transmitting of data for stage 2 joining. Clearly, to transmit the first data set 0, chains generated from image sections 0, 1, 2, and 3 must have been received.

The score chart was essential, due to the asynchronous and indeterminate nature of the tasks being performed. Just because four sets of Partial Boundary chains had been received by SUPPLY, during stage 1 for example, it could not be assumed that these were the chains generated from image sections 0, 1, 2, and 3, and it was observed that the sections were rarely returned to SUPPLY in numerical order.

When chains were received during a stage, their identities were used to mark the appropriate score chart as being received. When no Partial Boundary chains were generated from an image section, an empty set of chains was returned, and this used to mark the score chart. This was necessary to ensure that the data was marked as having been received, even though no chains had been generated.

9.8 Boundary Chain Code Implementation Results

The results for using discrete operations to perform parallel Boundary chain coding are shown in tables 9.1 and 9.2. The results obtained for the parallel implementation of the Boundary Chain coding are shown in tables 9.3, 9.4 and 9.5, using three data section sizes and an image size of 128 by 128. The test data used in these cases was BLOB, (shown in Appendix).

Number of Transputers	Execution Timings/mS Threshold	Binary Edge	Chain Code	Total
1	181	314	389	884
4	58	92	89	239
10	50	57	66	173
20	57	64	50	171
32	61	64	50	175

Table 9.1. Performance of Discrete Operations for Boundary Chain Coding (16 by 32 Data Section).

Number of Transputers	Execution Timing/mS	Performance Speed Up	Efficiency
1	884	1	100%
4	239	3.7	92.5%
10	173	5.1	51%
20	171	5.2	26%
32	175	5.1	16%

Table 9.2. Performance Obtained using Discrete Operations for Boundary Chain Coding (16 by 32 Data Section).

Number of Transputers	Execution Timing/mS	Performance Speed Up	Efficiency
1	797	1	100%
4	215	3.7	92.5%
10	87	9.2	92%
20	54	14.8	74%
32	52	15.3	48.8%

Table 9.3. Performance of Parallel Boundary Chain Coding (16 by 32 Data Section).

Number of Transputers	Execution Timing/mS	Performance Speed Up	Efficiency
1	878	1	100%
4	231	3.8	95%
10	94	9.3	93%
20	61	14.4	72%
32	44	19.9	62%

Table 9.4. Performance of Parallel Boundary Chain Coding (16 by 16 Data Section).

Number of Transputers	Execution Timing/mS	Performance Speed Up	Efficiency
1	969	1	100%
4	265	3.65	91%
10	108	9	90%
20	70	13.8	69%
32	60	16.2	50.6%

Table 9.5. Performance of Parallel Boundary Chain Coding (8 by 16 Data Section).

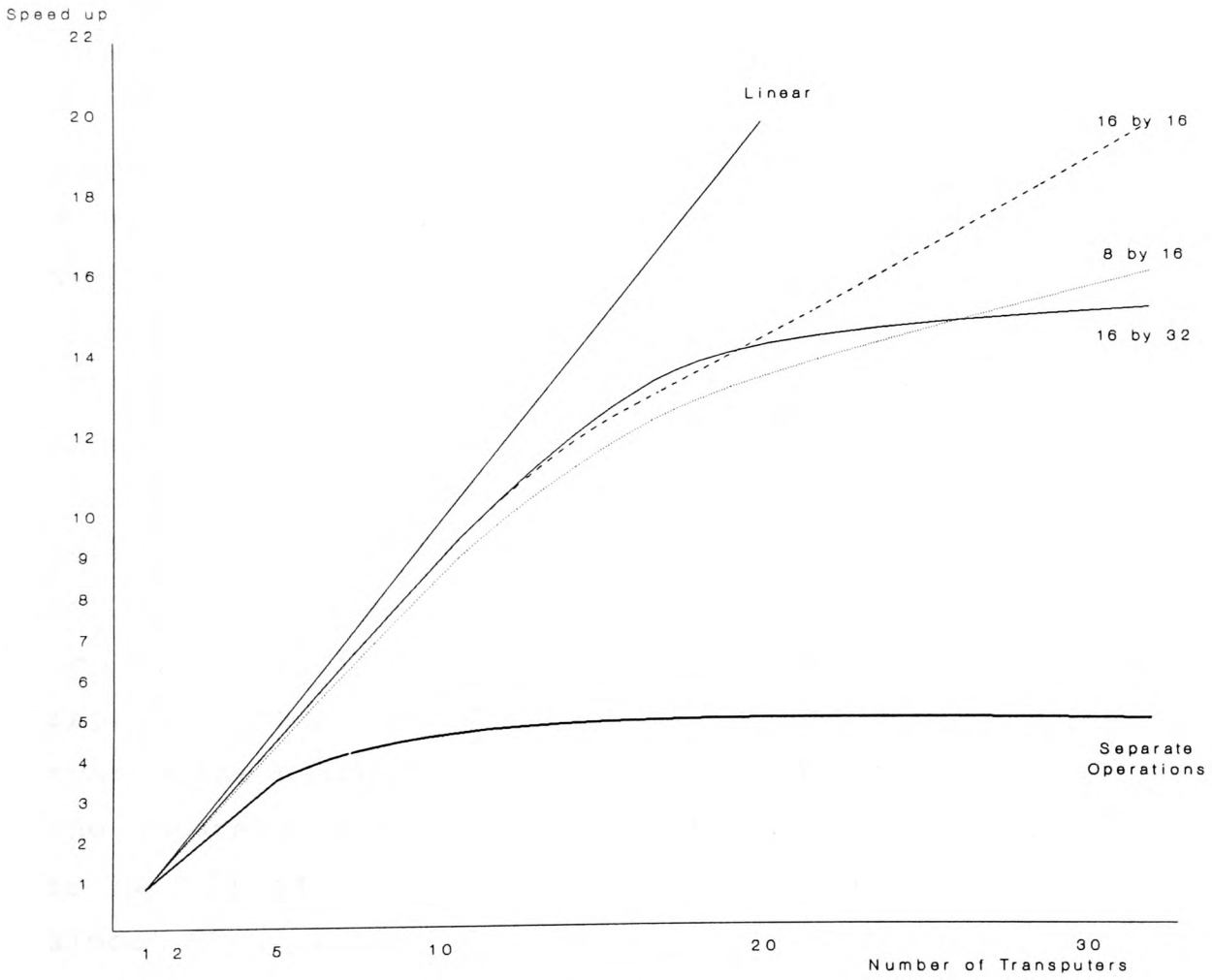


Figure 9.14. Performance of Parallel Boundary Chain Coding against Number of Transputers.

9.9 Discussion of Results

Discrete Operations. The results clearly show the benefits of using the SUPPLY and DEMAND software architecture to allow many transputers to work in parallel on the Boundary chain coding. When separate operations were utilised, involving Threshold, Binary Edge, and Chain Coding, executed by one transputer, the execution time was 884 mS (table 9.1). Implementing these operations on the tree network (using 20 transputers) reduced this figure to approximately 173 mS, a speed improvement of 5.1 times (Using a data size of 16 by 32) (table 9.2). The speed enhancement was not increased by using 32 transputers, however, due to the image section size used.

Section Size of 16 by 32. Combining these operations into a single operation yielded a far faster execution performance. The Boundary chain coding described here operated on grey level image sections. A Threshold operation was performed, then the improved chain coding algorithm operated on the binary object, without having to detect the edges first. This yielded comparative performance figures of 797 mS on one transputer, using a 16 by 32 data section, reducing to 54 mS on 20 transputers, and 52 mS on 32 transputers (table 9.3). These timings represent speed improvements of approximately 15 over one transputer using 32 processors, and 3.3 times faster than the separate operations technique. The speed up was found to be 90% of linear up to 13 transputers, reducing to almost 0% of linear from 14 to 32 transputers.

Section Size of 16 by 16. Altering the image section size affected the timings obtained, because the parallelism granularity was altered. Changing the image section size

to 16 by 16 allowed the parallel version of the Boundary chain coding to execute in 878 mS on one transputer, (table 9.4) reducing to approximately 60 mS and 43 mS on 20 and 32 transputers, representing speed ups of 14.4 and almost 20 respectively. The speed up was found to be 93% of linear with up to 13 transputers, reducing to 40% of linear using 14 to 32 processors.

Section Size of 8 by 16. Using an image section of size 8 by 16 made the parallelism granularity smaller, but required more distributed join up stages. Referring to table 9.5, the performance in this case was 969 mS on one transputer, 70 mS on 20 processors (a speed up of 13.8), and 60 mS on 32 transputers (a speed up of 16.2). The performance speed up was found to be 88% of linear up to 13 transputers, reducing to approximately 21% of linear using from 14 to 32 transputers.

9.10 Conclusions

The Boundary Chain Code of an object was generated by the multi-transputer system in parallel. Each image section sent to the DEMAND processes was Thresholded, then any edges present were coded into Chain Code sequences, using a look up table indexed by the groups of 3 by 3 pixels.

Partial Boundaries were then passed back to the SUPPLY process, where they were grouped for joining to others, generated from adjacent image sections. The distributed joining of the Partial Boundary chains was achieved in three stages when using a section size of 16 by 32. Chains from Four larger logical image sections were distributed together (figure 9.13), to allow DEMAND processes to test

start and end points to determine chains that would join.

Streamlining was employed to allow a degree of overlap of the four stages used in the distributed join up of the Partial Boundary chains. This allowed some DEMAND processes to start working on stage *N+1*, while others were still working on stage *N*.

Overall the boundary of the test object was coded into a single chain in 54 mS with 20 transputers, and 44 mS using 32 transputers. While the network does not yield perfectly linear speed up with increasing numbers of transputers, the techniques employed here allowed a speed up of 14.8 with 20 processors, and 19.9 with 32 processors. The speed up was found to be 93% of linear using up to 13 transputers, and 40% of linear from 14 to 32 transputers. The non linear speed up is due to the image data volume associated with each section during stage 1 (including neighbouring pixel values), when it was distributed for the initial Partial Boundary generation operation to be performed on it.

If a Digital Signal Processor (DSP) device was incorporated into the system, (as proposed in Chapter 13), the performance of the Boundary chain coding would be significantly improved. The use of more links per transputer would also allow higher order tree networks to be used, allowing a dramatic improvement in the overall performance. It is predicted that the speed up obtained using these strategies would allow the use of 50 - 100 transputers to be used, with the performance increases being approximately 80% - 90% of linear.

CHAPTER 10

PARALLEL IMPLEMENTATION OF CONVEX HULL

10.1 Introduction

The Convex Hull of a set of points is defined as the smallest superset such that all points on the boundary of the superset are convex. The Convex Hull of an object in an image can therefore be considered to be the smallest enclosing convex polygon, or the total region that would be enclosed by a rubber band stretched around the object (figure 10.1). All areas that are within the Convex Hull but not actually part of the object itself are termed Convex Hull Deficiencies. There are two forms of Convex Hull Deficiencies, areas totally enclosed by the object are termed Lake Deficiencies, and areas not enclosed by the object termed Bay Deficiencies (figure 10.2).

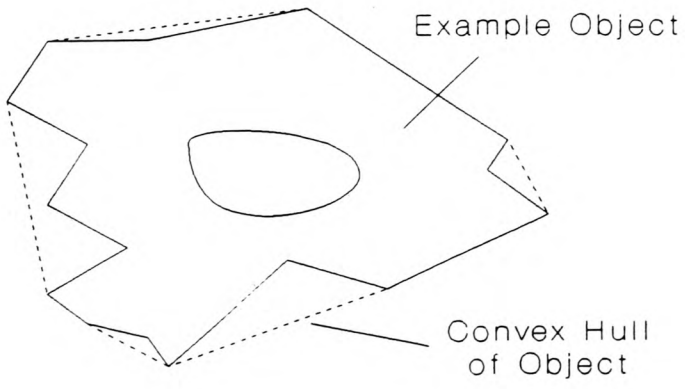


Figure 10.1. Example Of Convex Hull Formation.

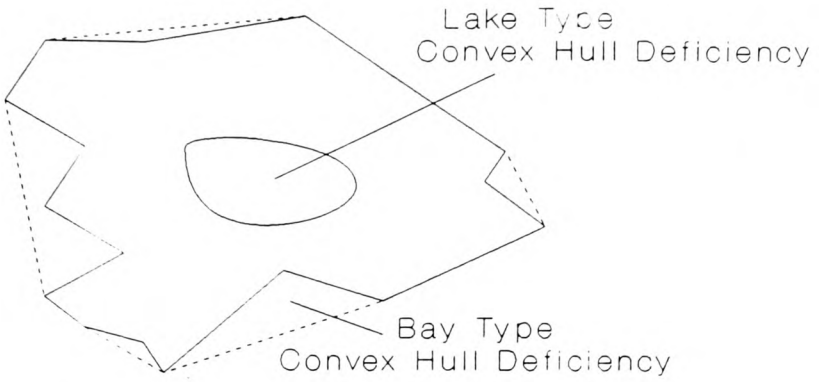


Figure 10.2. Convex Hull Deficiencies.

Several Convex Hull solutions have been proposed [143, 144]. Most of these solutions are not suitable for a parallel implementation, as they require access to the global image at many stages of the algorithm. Some of these algorithms also require complicated mathematics, involving trigonometric functions, thus rendering them less attractive.

10.2 A Solution in Image Space

A straightforward solution to the Convex Hull problem was realised in occam, and is shown in figure 10.3. Initially four points can be found that will always lie on the Convex Hull, the two *horizontal* and two *vertical* extremities (figure 10.4). In the program segment, these are (x.min, y.x.min), (x.max, y.x.max), (y.min, x.y.min) and (y.max, x.y.max). From each of these four points, a tangent to the object was scanned, to determine whether any other points

could be found. If an object intersection was found along the tangent, then this point was on the Convex Hull. The appropriate line was plotted to this point, then a tangent was scanned from it. If no other points were found, then a new tangent had to be scanned, which would be at an angle to the first.

```
PROC look.along.line (VAL INT x.start, y.start, x.end, y.end,
 VAL []BYTE image, BOOL found, INT x.value, y.value)
  ...   look along line for any white pixels
:

PROC plot.line (VAL INT x.start, y.start, x.end, y.end, []BYTE image)
  ...   plot white pixels along line
:

SEQ -- Main Convex Hull Process

  ...   get y.min and x at y.min, (x.y.min)
  ...   get y.max and x at y.max, (x.y.max)
  ...   get x.min and y at x.min, (y.x.min)
  ...   get x.max and y at x.max, (y.x.max)
  ...   assign start coordinates, x1 := x.y.min,  y1 := y.min

  -- 1st quadrant
  y.try := y.min -- trial values for end of tangent
  x.try := x.max
  ...  look.along.line (x1, y1, x.try, y.try, image.1, found, x, y)
  IF
    NOT found
      ...   adjust x.try and y.try
    else
      ...   plot.line (x1, y1, x, y, image.2)

  -- 2nd quadrant
  y.try := y.max -- trial values for end of tangent
  x.try := x.max
  ...  look.along.line (x1, y1, x.try, y.try, image.1, found, x, y)
  IF
    NOT found
      ...   adjust x.try and y.try
    else
      ...   plot.line (x1, y1, x, y, image.2)

  -- 3rd quadrant
  y.try := y.max -- trial values for end of tangent
  x.try := x.min
  ...  look.along.line (x1, y1, x.try, y.try, image.1, found, x, y)
  IF
    NOT found
      ...   adjust x.try and y.try
    else
      ...   plot.line (x1, y1, x, y, image.2)

  -- 4th quadrant
  y.try := y.min -- trial values for end of tangent
  x.try := x.min
  ...  look.along.line (x1, y1, x.try, y.try, image.1, found, x, y)
  IF
    NOT found
      ...   adjust x.try and y.try
    else
      ...   plot.line (x1, y1, x, y, image.2)
```

Figure 10.6. Outline of Convex Hull Algorithm in Image Space.

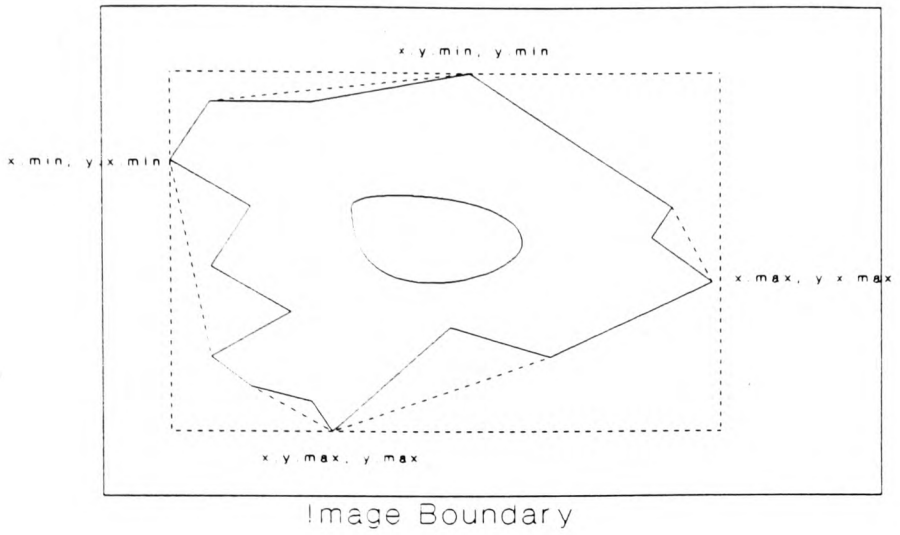


Figure 10.4. Four Initial Points on the Convex Hull.

The image space that was scanned was limited to the rectangle bounded by the horizontal and vertical extremities (figure 10.4). While numerous short cuts were employed in this computation, in order to enhance the performance, the execution was still lengthy and time consuming. Moreover, a parallel implementation would not be easy. Clearly four parallel computational tasks can be identified, these being the four quadrants in the image. These could be processed by four different processors, in parallel, but it would not be straightforward to further divide the four quadrants into more concurrent tasks, for further performance gains. Table 10.1 shows the performance timing for the Convex Hull in Image Space, executing on one transputer.

Test Data	Execution Timing/mS
BLOB	613

Table 10.1. Performance of Convex Hull Algorithm in Image Space, on one Transputer.

The test data used was the test figure, BLOB, shown in the Appendix. This algorithm operated on an object in image space, and produced the boundary of the Convex Hull as a line plotted by white pixels, also in image space. To produce the chain code of the Convex Hull and the chain code of the boundary, a separate chain coding algorithm had to be executed. The chain coding algorithm timings are shown in table 10.2, and include the timings shown in table 10.1. The chain coding algorithms were executed on one transputer, in this case, in order to compare this realisation with the parallel Convex Hull in chain code space, discussed in this Chapter.

Test Data	Execution Timing / mS		
	Chain Code Boundary	Chain Code Convex Hull	Total including Convex Hull
BLOB	389	200	1202

Table 10.2. Adding Chain Coding Subsequent to Convex Hull in Image Space, on one Transputer.

10.3 A Solution in Chain Code Space

The chain code of the Convex Hull of an object can be calculated from the chain code of its Boundary [145]. This algorithm has the advantage of utilising the low data volume inherent in chains, and avoids any significant mathematics, thus yielding an efficient realisation.

The Wilson algorithm initially works through the chain direction codes, selecting those which may lie on the convex hull. As the object is coded clockwise, any right turns made by two consecutive direction codes indicate a local point of convexity. Using the standard 8 direction codes, and if *D1* and *D2* are two consecutive direction codes, then for local convexity:

```
        D1 = ((D2 + 1) \ 8)
or      D1 = ((D2 + 2) \ 8)
or      D1 = ((D2 + 3) \ 8)
```

(where \ represents Modulo)

Any local points of convexity found in the chain are marked as possible points on the Convex Hull.

The Three Point Test. Three consecutive points of local convexity are selected, and a test is performed using their X and Y coordinates to determine whether the middle point is on the object side or the non-object side of an imaginary straight line joining the two other points (figure 10.5). The chain of a boundary was coded clockwise.

Referring to figure 10.5(a), if the middle point *B* is on the object side of the straight line joining the two outer

points *A* and *C*, then it is discarded, because it is therefore inside the Partial Convex Hull between the two outer points. If it is on the non-object side, however, as in figure 10.5(b), it must be retained, as it could be on the Convex Hull.

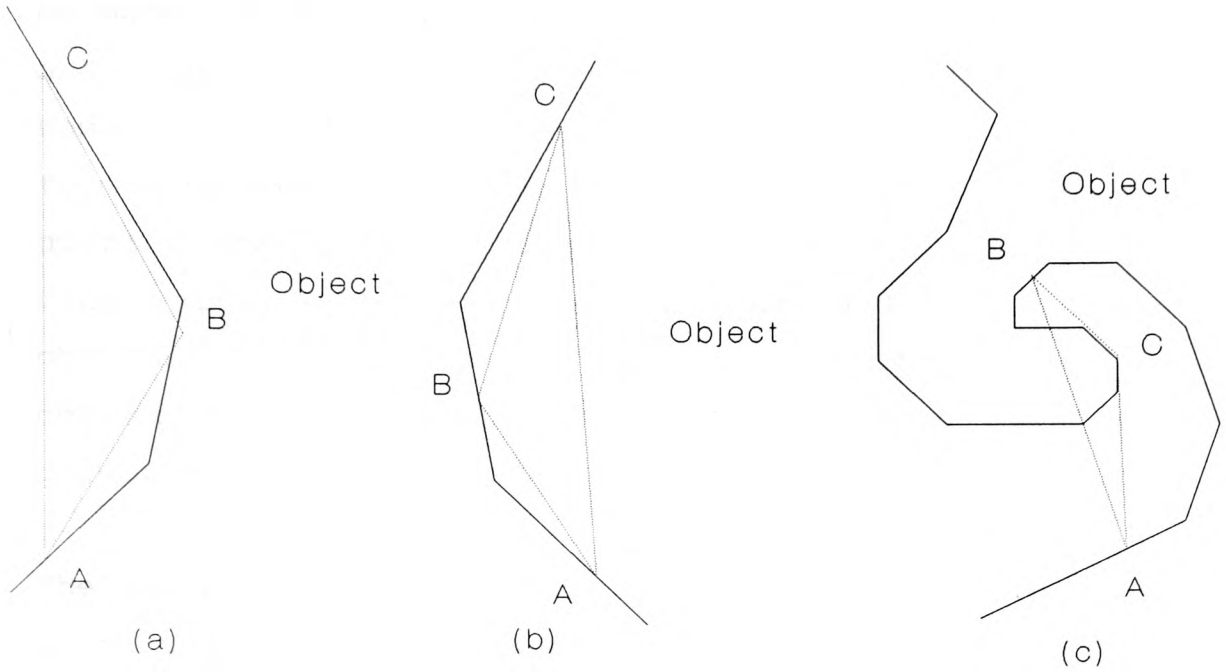


Figure 10.5. Computation of Partial Convex Hull from Partial Boundary.

When these three points have been tested, the first point is marked, and the next three points of local convexity are tested. In this way the entire chain is traversed. The procedure is iterated, until no points have been discarded. The Convex Hull has then been obtained for the chain, defined by the remaining possible points of convexity. Straight lines between these points yield the overall Convex Hull boundary of the object.

10.4 Improvements to the Chain Code Solution

When the initial algorithm had been realised in occam and tested on test data, several cases arose which caused the algorithm to fail. The three point test procedure had to be more complicated than was at first apparent [146]. Not only must it work in any of the four basic quadrants, but some combinations of local points of convexity resulting from a spiral shaped part of an object can make the middle point appear to be on the object side when it is not, or vice versa. In figure 10.5(c), point B is on the non-object side of line AC, which would imply that it is convex, but in actual fact it must be discarded, as the object edge spirals around it.

The failure of the initial algorithm caused the whole three point test to be reexamined by Wilson, with some assistance from the Author [147]. This reexamination led to the basic algorithm being proved, and made robust, with a new sub test being appended.

10.5 A Parallel Solution in Chain Code Space

Using a modified version of the Wilson algorithm, it was possible to generate Partial Convex Hull (PCH) chains from Partial Boundary (PB) chains, in parallel, throughout the multi-transputer configuration [146]. The Partial Boundary chain generation was described in Chapter 9. The parallel Convex Hull computation also generated the complete boundary of an object.

Partial Convex Hull Chains. A Partial Convex Hull chain includes useful information for subsequent Convex Hull

calculations. Typically a Partial Convex Hull chain consists of a few defining points, which mark the convex points on the chain. Other points in between these defining points are not considered by subsequent Convex Hull operations, as they are only present to form a line between the defining points.

Joining Two Partial Convex Hull Chains. When a Partial Boundary chain was joined to another Partial Boundary chain, a new Partial Convex Hull chain had to be computed. The new Partial Convex Hull chain was calculated from the two original Partial Convex Hull chains as shown in Figure 10.6. This computation was greatly facilitated by the use of the original Partial Convex Hull chains, because these chains already partially defined the target chain. This approach was implemented within each of the DEMAND processes, yielding a parallel Convex Hull solution.

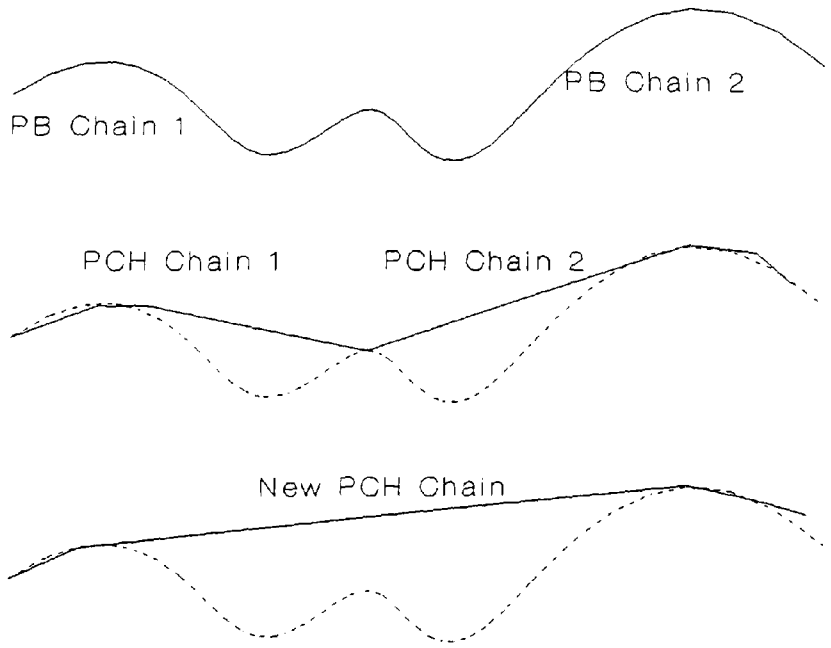


Figure 10.6. A New Partial Convex Hull Chain from Two Partial Convex Hull Chains.

Parallel Convex Hull Stages. The computation of the Convex Hull was implemented in several stages due to the image sectioning carried out for the parallel computations, as with the Boundary generation in Chapter 9. The number of stages depended upon the size of the image sectioning, with image sections of 16 by 32 requiring four stages. The test figure, BLOB, is shown at the end of each stage, to illustrate the procedure when using a 16 by 32 data section.

Stage 1. The first stage consisted of image sections being distributed to the DEMAND processes, as with low level operations, with neighbouring pixel values included. Partial Boundary chains were produced locally for any object edges present in the sections. When a chain representing an edge had been produced, the associated Partial Convex Hull chain was also computed. Start and end points of Partial Boundary chains were, by definition, on the Partial Convex Hull. Pairs of Partial Boundary and Partial Convex Hull chains were thus produced, and sent back to SUPPLY.

Pairs of chains received by SUPPLY were stored in arrays using the chain identity as the index. This facilitated the selective retrieval of specific chains for use in subsequent stages. An image representation of the test object, BLOB, at the end of stage 1 is shown in figure 10.7. All work at this stage was performed on chains, and not two dimensional data. It can be seen that the Partial Convex Hulls have been formed within the image sections.

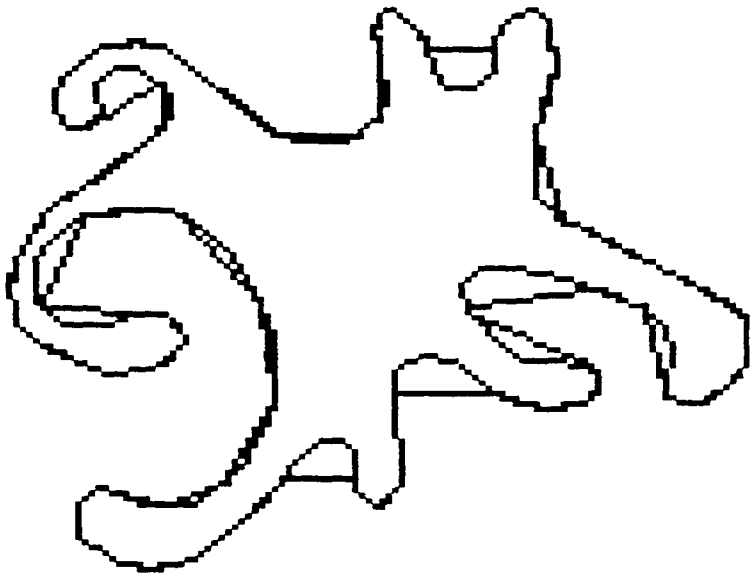


Figure 10.7. Test Object, BLOB, at the end of Stage 1.

Stage 2. When all image sections had been processed for stage 1, Partial Boundary chains must be tested for joining together to form larger Partial Boundary chains.

To send out a stage 2 group of chains, the values of the four stage 1 identities were calculated, using the formulae shown in figure 9.12, and those chain pairs copied from the stage 1 array. The chains were then transmitted, tested for joining, and joined where possible in the DEMAND processes.

Whenever any Boundary chains were thus joined, the new Partial Convex Hull chain must also be produced, from the respective Partial Convex Hull chains, as before. If a boundary chain was found to be closed, then the complete Convex Hull chain was produced from the joined Partial Convex Hull chains.

The test object at the end of stage 2 is shown in figure 10.8. Again, it can be seen that the Partial Convex Hulls have been formed within the limits of the larger

logical image sections. Stage 2 chains were again stored within the SUPPLY process in a two-dimensional array, this time using the second stage logical image section identity as the index.

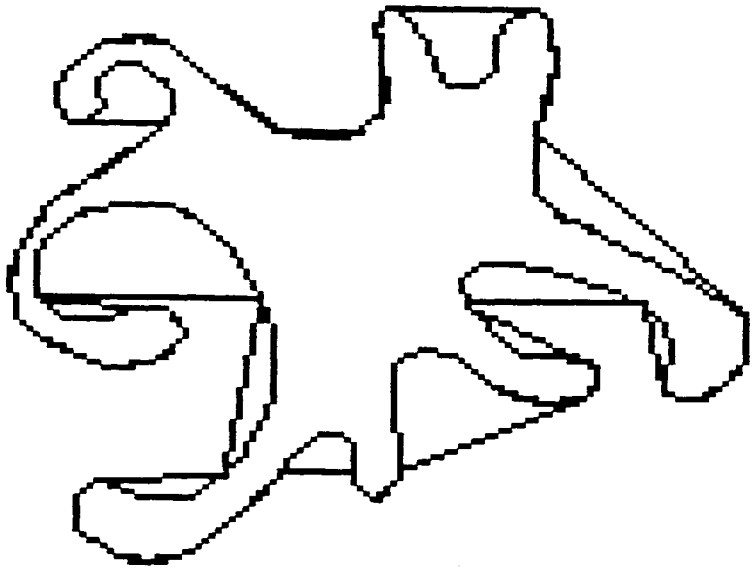


Figure 10.8. Test Object, BLOB, at the end of Stage 2.

Stage 3. Stage 3 consisted of grouping together sets of stage 2 chains produced from adjacent stage 2 sections, so that four stage 2 groups were sent out for possible joining. The same formula used in stage 2, with different parameters, gave the four stage 2 identities for a given stage 3 identity. During this stage chains from the two halves of the image were now being tested for joinability, in parallel. The test object on conclusion of stage 3 is shown in figure 10.9. It can be seen that the overall Convex Hull has now started to be formed, with almost the correct Convex Hull formed in each of the two halves of the image.

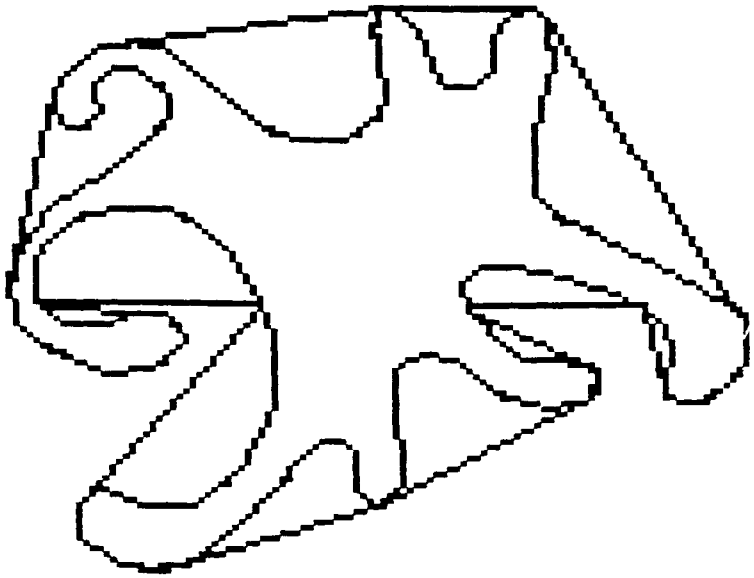


Figure 10.9. Test Object, BLOB, at the end of Stage 3.

Stage 4. When stage 3 had completed, the root SUPPLY process had to perform the final stage (stage 4). The two stage 3 groups of chains, formed from the two halves of the image, were tested together for joinability, and the appropriate Partial Convex Hull chains computed where required. The test object is shown in figure 10.10, at the conclusion of the final stage.

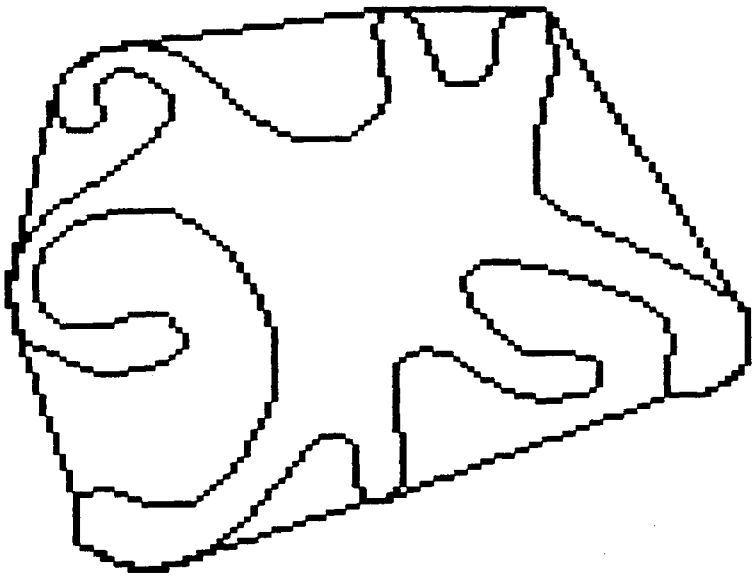


Figure 10.10. Test Object, BLOB, at the end of Stage 4.

10.6 Streamlining of Distributed Convex Hull Stages

The stages used in the distributed Convex Hull formation were overlapped, using the Streamlining technique.

The implementation of Streamlining during the four stages was essentially similar to that employed for the parallel Boundary chain code generation. While some DEMAND processes were computing the Partial Boundary and Partial Convex Hull chains associated with an image section for stage 1, other DEMAND processes were joining Partial Boundary chains, and computing the new Partial Convex Hull associated with four groups of stage 1 chains, for stage 2.

10.7 Parallel Convex Hull Implementation Results

The results obtained for the parallel implementation of the Convex Hull are shown in tables 10.3, 10.4 and 10.5, using three different section sizes, and an image of size 128 by 128.

Number of Transputers	Execution Timing/mS	Performance Speed Up	Efficiency
1	1513	1	100%
4	420	3.6	90%
10	166	9.1	91%
20	92	16.4	82%
32	60	25.2	78.8%

Table 10.3. Parallel Convex Hull Implementation Results (Using a 16 by 32 Data Section).

Number of Transputers	Execution Timing/mS	Performance Speed Up	Efficiency
1	1660	1	100%
4	461	3.6	90%
10	182	9.1	91%
20	92	18	90%
32	61	27.2	85%

Table 10.4. Parallel Convex Hull Implementation Results (Using a 16 by 16 Data Section).

Number of Transputers	Execution Timing/mS	Performance Speed Up	Efficiency
1	1840	1	100%
4	512	3.6	90%
10	214	8.6	86%
20	117	15.7	78.5%
32	81	22.7	71%

Table 10.5. Parallel Convex Hull Implementation Results (Using a 8 by 16 Data Section).

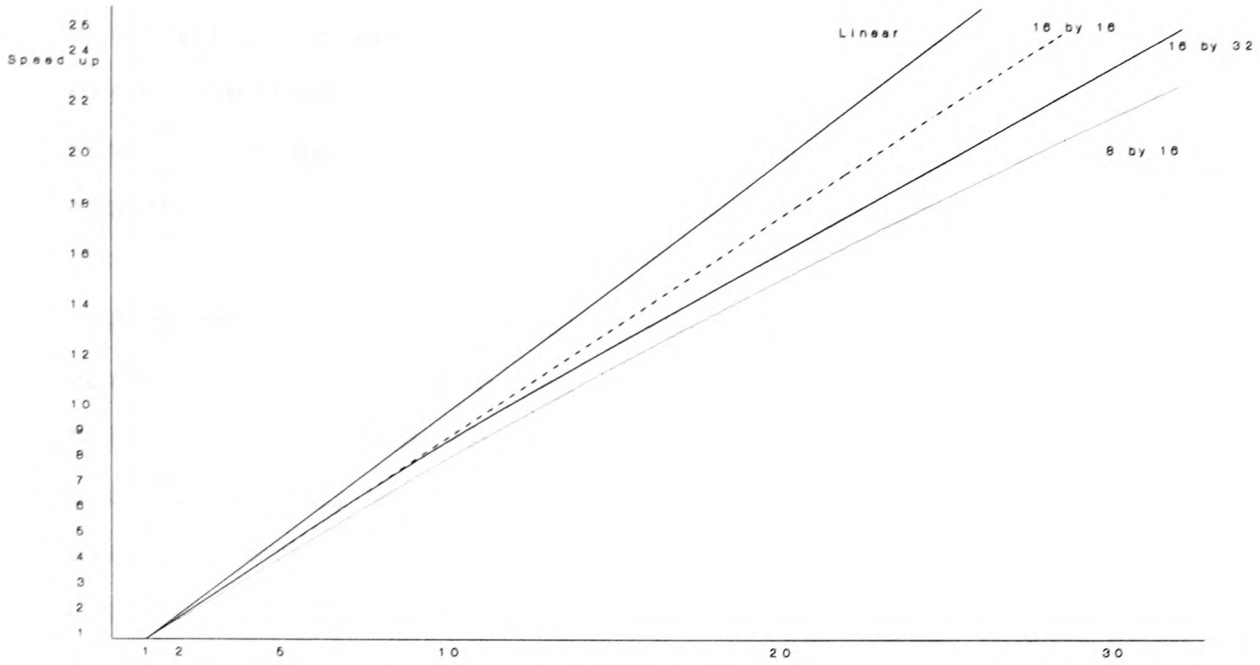


Figure 10.11. Performance of Parallel Convex Hull against Number of Transputers.

10.8 Discussion of Results

The sequential image space solution to the Convex Hull, using one transputer yielded a performance timing of 613 mS. This did not use any data division techniques. Adding the timings for chain coding the Boundary and Convex Hull chains made the total to be 1202 mS (table 10.2). Using the parallel implementation on one transputer, (table 10.3) the execution timing was obtained to be 1513 mS with a 16 by 32 data section. Using a 16 by 16 data section (table 10.4) the timing was 1660 mS, and 8 by 16 was 1840 mS (table 10.5). The parallel versions thus yielded execution timings approximately 1.3, 1.4, and 1.5 times slower than the sequential version, on one transputer. This is not surprising, as the image space solution executed on one transputer with no data division

or work distribution. The parallel implementations, when executed on one transputer, however, caused data to be divided into sections, and *distributed to itself*. This explains why, using different data section sizes, the resultant performance varied on one processor, and was always slower than the image space algorithm.

The parallel implementation could be (and was) executed on more than one transputer, however, and the results prove that the performance was greatly enhanced. Timings of 166 mS, 92 mS, and 60 mS were obtained using 10, 20, and 32 transputers respectively, and a 16 by 32 data section (table 10.3) These represent speed improvements of 9.1, 16.4 and 25.2 over the parallel version on one transputer, and 7.2, 13.1 and 20 times faster than the single transputer image space solution.

Using a 16 by 16 section, the performance increases were 9.1, 18, and 27.2 over one transputer (table 10.4), using 10, 20 and 32 processors, and 6.6, 13.1, and 19.7 times faster than the image space algorithm.

Using a 8 by 16 data section, the increases in performance were 8.6, 15.7, and 22.7 using 10, 20, and 32 transputers over using one transputer (table 10.5), and 5.6, 10.3, and 14.8 times faster than the image space solution on one transputer.

The speed ups obtained (shown in figure 10.11) were found to be approximately 87% of linear using a 16 by 32 data section, 90% with a 16 by 16 section, and 80% using an 8 by 16 section, and up to approximately 13 transputers. Above this number of processors, the speed ups obtained were reduced slightly, to 71% for the 16 by 32 data section, 83%

for 16 by 16, and 57% for 8 by 16. This was when the fourth layer of nodes was used in the tree network.

10.9 Conclusions

A novel method of arriving at the Convex Hull of an object using a multi-processor system has been illustrated. The Convex Hull was formed from the Boundary Chain Codes. After a Partial Boundary had been generated, its Partial Convex Hull was computed. As Partial Boundaries were joined to others, their combined Partial Convex Hull was also computed, from their respective Partial Convex Hulls. In this way, a complete Convex Hull was determined.

The stages involved with this implementation were overlapped, to allow some DEMAND processes to work on the last sections of data from one stage, while, in parallel, other DEMAND processes worked on the first groups of chains from the next stage.

The results obtained, using three different image section sizes, indicate the suitability of this software and hardware architecture for performing more complex and non determinate algorithms. The speed ups using a 16 by 16 data section were found to be 90% of linear up to 13 transputers, and 83% using up to 32 processors.

The use of higher order trees, and a front end Digital Signal Processor device, would allow many more processors to be used, without significant performance degradations. It is predicted that a speed up of 90% of linear could be achieved using 50 to 100 transputers, when performing algorithms such as the one discussed in this Chapter.

CHAPTER 11

FEATURE VECTOR COMPUTATION

11.1 Introduction

In order to recognise and classify an object within an image, features must be extracted that characterise the object. Such features could, in the most simple case, indicate the presence or absence of an object. More complicated features involve attributes and measures, to enable one object class to be distinguished from another.

Features were extracted from an object's silhouette boundary shape, and from the Convex Hull of that shape. The features used included the Shape Factor of the object and its Convex Hull, the number and type of the Convex Hull Deficiencies, and their area ratios to the area of the object. A Feature Vector was constructed, comprised of a sequence of numbers representing these feature measures.

11.2 Convex Hull Deficiencies

Algorithms for generating Bay and Lake Convex Hull Deficiencies were implemented using the SUPPLY and DEMAND software architecture.

Bay Convex Hull Deficiencies. The chains representing Bay type Convex Hull Deficiencies (CHDs) were computed from the object boundary and Convex Hull chains directly. Bay CHDs

were found by following the object and Convex Hull perimeter chains between their intersections. Special attention had to be paid to avoid generating spurious Bay CHDs where the two chains were almost (but not quite) coincident.

Lake Convex Hull Deficiencies. Lake CHDs were treated as separate objects during the initial Object Boundary chain coding, as they do not have any interaction with the boundary, by definition. These Lake CHD chains were thus computed concurrently with object boundary chains. The software could easily be extended to produce a shape describing concavity tree, which classifies a boundary shape according to its Convex Hull Deficiencies, the CHDs of each of the CHDs, and so on.

11.3 Areas and Perimeter Lengths

Areas and perimeter lengths were directly computed from the chain codes, in the normal way, by integration along the x-axis [142]. A modified form of the technique proposed in [9] for Area computation from a chain code was devised that correctly allowed for the edge pixels of an object. The edge pixels of an object must be considered to be part of the object area. Otherwise the area computed will be less than the actual area by a factor proportional to the number of pixels on the perimeter. Using the standard chain code directions (shown for convenience in figure 11.1), the area variations for each direction code are shown in table 11.1.

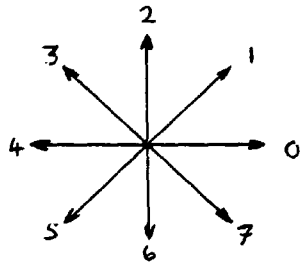


Figure 11.1. Chain Code Direction Codes.

Chain Code	Delta A	Delta B	Delta C
0	-B	0	1
1	-B	-1	√2
2	+1	-1	1
3	+B+1	-1	√2
4	+B+1	0	1
5	+B+1	+1	√2
6	0	+1	1
7	-B	+1	√2

Table 11.1. Computation of Area from Chain Codes.

Algorithm. A variable *B* is added to or subtracted from the area accumulated, according to each direction code in the chain being traversed. The value *B* is initially set to zero. This keeps the value of B minimal, so that any errors introduced by an unbalanced addition or subtraction of B are minimised. Movements from *left* to *right* cause the value of B to be subtracted from the area. Movements from right to left cause the value (B + 1) to be added to the area. Movements vertically down increase B by 1, and movements vertically up decrease B by 1.

Description of Area Computation. Referring to figure 11.2, movements from *left* to *right* must cause the area to be altered by -B. This cannot be inverted for movements from

right to left, however, because that would cause the top edge to be subtracted from the area. Movements from right to left must therefore add (B + 1). Vertical movements must cause the area to change by 1. If this is not performed, the area will be too low by an amount equal to the number of pixels on the vertical sides of the object. If this is performed for both vertically up and down movements, however, then the area computation will be too high by an amount equal to half the number of pixels on the vertical sides. One vertical direction only must add 1 to the area, either vertically up or down. It was found convenient to add 1 when moving vertically up, and not changing the area when moving vertically down.

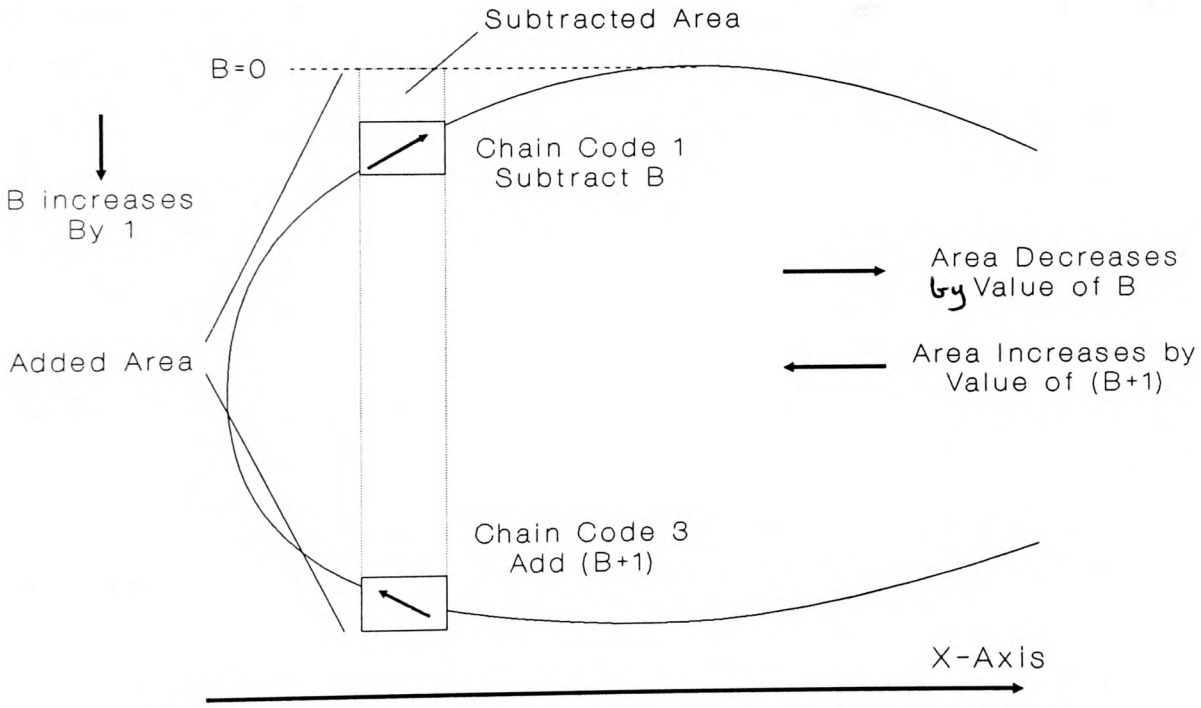


Figure 11.2. Area Computation by Integration Along X-Axis.

Perimeter Computation. The perimeter is altered according to the value of the variable *C*, with C being added each time. The value contained in C varies according to table 11.1, being 1 for direction codes 0, 2, 4 and 6, and $2^{1/2}$

for diagonal codes 1, 3, 5 and 7. The perimeter may also be calculated from the number of *odd* and *even* chain code directions present. The number of even codes is added to ($2^{1/2}$ * the number of odd codes).

Parallel Implementation. The software architecture distributed closed chains of Object Boundaries, Object Convex Hulls, and Convex Hull Deficiencies for concurrent computation of areas and perimeters. These features of a closed chain would typically be calculated by different processors.

When there were few closed chains obtained from an image, the system had the flexibility for load distribution to allow computation of *partial* areas and perimeter lengths from parts of chains. This gave a better spread of work than if, say, only one processor performed the area computation and another calculated the perimeter. Global figures for each chain were obtained by the root SUPPLY process summing the partial areas and perimeters. This allowed the utilisation of the multi-processor configuration to be kept high.

11.4 Feature Vector

A full length Feature Vector [148] was used initially, having 5 component attribute types. These comprised the Shape Factor of the Object itself (F_O), which is given by :

$F_O := (P * P) / A$

(where *P* is the perimeter, and *A* the area)

the number and type of the Bay and Lake Convex Hull

Deficiencies (N_b and N_l), their Shape Factors (F_i) and the ratios of their areas to the Object area (R_i),

(where N_b is the number of Bay CHDs, N_l is the number of Lake CHDs, and i = 1, 2, 3 . . . (N_b + N_l))

In practice, however, it was found that this Feature Vector could be reduced substantially, by omitting the details of any smaller Convex Hull Deficiencies. As an example, those CHDs with an area ratio to that of the Object of less than 1/10 could be omitted.

The Feature Vector components were calculated in parallel, with the root process controlling and distributing the relevant parameters and data for the calculations to be performed.

11.5 Conclusions

A Feature Vector was constructed, including object boundary and Convex Hull Shape Factors, Convex Hull Deficiencies, and area ratios. The Feature Vector enabled a two dimensional object within an image to be described using a sequence of numbers.

Bay type Convex Hull Deficiencies were computed from the intersections of the complete Boundary and Convex Hull Chains. Lake Deficiencies were computed concurrently with the Object Boundary, being treated as separate objects, except they were coded anti-clockwise, and not clockwise, as with Objects.

Areas and Perimeters were calculated from the Chain Code

Sequences. Where few long chains were present, partial chains could be distributed, to allow a greater degree of parallelism to be obtained. Shape Factors were then calculated, to produce the Feature Vector.

Once a Feature Vector has been computed for an object, it can be used to determine the amount of difference of that object from previously generated Feature Vectors. This is discussed in Chapter 12.

CHAPTER 12

IMPLEMENTATION OF SCENE INTERPRETATION

12.1 Introduction

The scene interpretation algorithms, including simple *Learn* and *Recognise* procedures based on the Feature Vector [149, 150] were devised but not fully implemented and characterised. Early results proved most promising, however, with distance measures being calculated very quickly by the system.

12.2 Learn

The *Learn* procedure consisted of finding the Feature Vector comprising of several different measures obtained from the Object. The Feature Vector thus generated could be added to a table, or database, which would be stored on disk in the Host Development System. The table structure devised involved vectors of variable length stored in an array. The table would be sorted on the first 2 attributes, the number of Lake and Bay C.H.D.s. This would enable a preliminary search within the Recognise procedure to establish the approximate place to start the distance measure calculations, within the Feature table.

Whenever a Feature Vector was to be added to the table, it had to be sorted and inserted at the correct place, depending upon the number of Lakes and Bays, and then the

whole feature table would be written back to disk.

12.3 Recognise

The *Recognise* process devised performed a nearest neighbour cluster analysis utilising a Euclidean distance measure [148]. The Euclidean distance *D* between two feature vectors can be calculated by:

$$D\ (F_1,\ F_2) := \left[\sum_{i=1}^{i=q} \mid F_1(i) - F_2(i) \mid^{2} \right]^{1/2}$$

Where D $(F_1,\ F_2)$ is the Euclidean distance between Feature Vectors F_1 and F_2, and *q* is the dimension of the Feature Vector.

In practice, this can be simplified by using a *squared* Euclidean distance measure. Using a Feature Vector with, say, 3 dimensions:

F_1 = A, B, C

then the squared Euclidean distance D^2 to another Feature Vector

F_2 = x, y, z

is given by

$$D_{12}^{2} = [\ (A - x)^2 + (B - y)^2 + (C - z)^2\ ]$$

In this application, Feature Space weightings (λ_i) were added to the individual Feature Vector component differences. The Squared Euclidean distance was therefore given by :

$$D_{12}^2 = [\ \lambda_1 (A - x)^2 + \lambda_2 (B - y)^2 + \lambda_3 (C - z)^2\ ]$$

This ensured that important dissimilarities between Feature Vectors were assigned higher numerical values than trivial differences. Differences such as the numbers of Lake and Bay Convex Hull Deficiencies were considered to be far more important than small differences in area ratios, or small differences in Shape Factors, for example.

$Object_1$ is exactly matched with $object_2$ if:

$$D_{12}^2 = 0$$

Typically, however, $D_{12}^2 <> 0$ so an exact match will not be obtained between these two vectors. The closest match must therefore be obtained, between Feature Vectors.

A Feature table has, say, *M* Feature Vectors. Generally many Vectors' distances need to be calculated from an object Vector in order to obtain the best match, i.e. where the value of D_{min}^2 is minimised.

$$D_i^2 = D\ (\ F,\ F_i\ )$$
$$D_{min}^2 = \text{MINIMUM}\ (\ D_i^2\ ) \qquad \text{where } i = 1,2,3...M$$

This is an ideal application for parallel processing. A part of the Feature database would be distributed to each DEMAND process, along with the Feature Vector to be matched. Each DEMAND process would compute the squared

distance from this Vector to each other Vector in its local Feature table. The local best matches,

$D_{min(j)}$ where j = 1,2,3...P (number.of.processors)

were then passed back to the SUPPLY process.
These *P* values were then compared, and the minimum was selected, yielding the pre-encountered object that was the closest match to the object under examination.

The classification method implemented involved computational complexity proportional to *N* * *M*, using an N-dimensional object describing mechanism, and M pre-learnt objects.

12.4 Conclusions

The implementation of simple forms of *Learn* and *Recognise* functions has been proposed and discussed. These algorithms have not actually been implemented fully, but preliminary tests showed promise, and verified the theory and the software devised.

Using a Feature Vector determined from Convex Hull Deficiencies, Shape Factors, and Area ratios, a topological, position and size independent object shape signature can be obtained. This signature was found to be essentially the same for an object as for the object's rotations, and a smaller copy of the object.

Learn comprised the extraction of feature information from an image, and the building of a Feature Vector that essentially described the shape of any objects in an image.

This can then be added to a table stored in the host system (or some type of non-volatile storage in a target application system).

The *Recognise* task built a Feature Vector, as with *Learn*, and computed the dissimilarity of this object from all other objects previously encountered. This was performed by calculating the Squared Euclidean Distance from this vector to all other vectors. The shortest distance would, therefore, indicate the best match of object to object class.

CHAPTER 13

FUTURE WORK

13.1 Front End Digital Signal Processor Device

A front end Digital Signal Processing (DSP) device could be used in between the Camera Digitiser and the tree network (figure 13.1). This device would perform many of the low level image processing transformations at high speed, including Enhancement, Thresholding, and Edge Detection. Ideally the DSP device would only pass chain coded edges to the transputer system, for further detailed analysis.

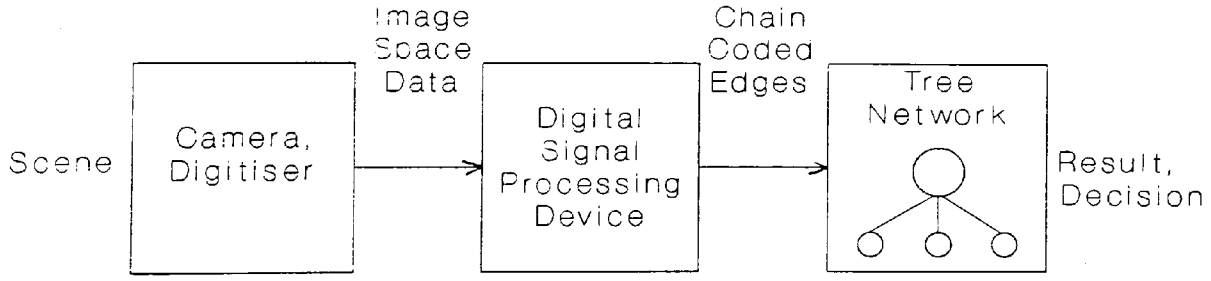


Figure 13.1. Front End DSP Device.

This would allow the transputer network to deal with the more complex Feature Extraction and Object Recognition algorithms, without having to perform the data intensive, low level functions, enabling a significant increase in the overall efficiency of the system.

13.2 Defect Detection

The system described could be utilised for Defect Detection within manufactured items in an Automatic Inspection cell. Items presented on a conveyor belt, for example, could be reduced to several features, and compared with an ideal "model" item. Alternatively, if only certain defects were to be encountered or checked for, then dedicated algorithms could be devised that would only check for those particular features. Such features might include cracks in glass or porcelain items, incorrectly shaped food items (biscuits, chocolates), or non edible articles in food (metal or bone fragments). The recent spate of glass and metal fragments appearing in certain baby foods highlights the need for this type of inspection.

13.3 Intelligent Syntactic Recognition

The system could be adapted to reduce an object to its constituent parts, in order to construct a more intelligent, higher level object description. Any lines present on the object would be noted, whether straight or curved, and built into larger primitive shapes. Squares, rectangles, circles, and triangles could be found, and their connectivity and relative positions obtained. Continuing this procedure could result in three dimensional sub-objects being built up, which would lead to the overall object under study being identified.

13.4 Robotic Vision

Some of the techniques proposed in this work could be used

to provide simple Robotic Vision [151]. Information including size, position, centroid, and shape could be obtained, and used to perform various robotic operations. Positional information might enable a robotic arm or gripper to move to the correct place to pick up an object, or allow a mobile robotic vehicle to track a path or road [152, 153].

13.5 Automated Assembly

Simple robotic vision using techniques proposed in this work could be used to enable an automated assembly cell consisting of a robotic arm and associated devices to adaptively pick and place items. It has been demonstrated that it is a straightforward task to teach a robotic system to pick and place items that are in predetermined positions and places [154], but the addition of simple computer vision techniques would allow a much more intelligent and adaptive approach to this.

13.6 Summary

A Front End Digital Signal Processing device, placed in between the camera digitiser and the multi-transputer network could be used to perform many of the low level image processing Enhancement, Thresholding, and Edge Detection. Either Edge Detected images or sequences of chain coded edges could then be passed to the processor network, for subsequent higher level algorithms.

The techniques featured in this work could be utilised to provide a variety of vision functions. Automated

inspection and defect detection, and Syntactic object recognition could be achieved. If used in conjunction with Robotics, simple robotic vehicle guidance, or Automated Assembly may be performed.

CHAPTER 14

CONCLUSIONS

14.1 Aim Of Research

It was the aim of the research to investigate the implementation of image processing algorithms upon a suitable hardware and software architecture. The use of many programmable devices in parallel is essential, in order to obtain the required flexibility and computational power necessary with higher order image processing algorithms.

The transputer is a powerful new 32-bit programmable microprocessor, designed with the parallel processing language occam, to allow multi-processor networks to be built, and the concurrent software to be specified and correctly executed. The use of the transputer for higher level image processing functions combines the requisites of a programmable, flexible and parallel processing device.

The research aimed to investigate the exploitation of a multi-transputer configuration to perform useful image processing algorithms efficiently. The work was also to investigate whether the execution of such algorithms could benefit significantly from using multi-transputer systems, in order that complex vision dependent tasks will be achievable having a realistic and practical performance.

14.2 Other Work and General Observations

Low level image processing transformations can be performed at high speed by hardware logic, VLSI, DSP devices, and SIMD arrays. Configurations featured in the literature for image processing applications have involved pipelines, arrays, and pyramids, using conventional processors as well as transputers. Mainly static load balancing strategies have been adopted with these topologies. Such algorithms are not data dependent, and consist of a large number of simple repetitive operations, performed over the entire image. These strategies cannot be used to perform complex feature extraction and scene interpretation tasks, however, as flexible programmable components are required. It is desirable for many such devices to execute concurrently, in order to enhance the execution speed of the overall task.

Very few systems have attempted the implementation of image processing algorithms other than low level transformations. Indeed, the author is not aware of one commercial system that utilises parallel processors in an MIMD mode of operation to perform any significant non data dependent algorithms.

Clearly much work remains to be done, so that the full capability and potential of such powerful configurations can be exploited to perform complex image processing algorithms. Not until this has been achieved can practical and useful vision dependent tasks, such as Intelligent Vision, Scene Analysis, and Mobile Robot Guidance, be realised.

14.3 Hardware and Software Designed for the Research

Camera and Monitor Interface. An interface was designed and constructed to allow a transputer to dynamically receive images from a camera digitiser, and display images on a video monitor.

Quad Transputer Board. A quad transputer board was designed and several were constructed for use in the research. This allowed the board design to be exactly tailored to offer the features and flexibility that were required for the investigative work carried out. Each transputer on the board had 256 Kbytes of 4 cycle interface DRAM, and terminated and buffered off board links. One example of this design tailoring was the routing of all four communication links to the edge connector for each of the four transputers. Other commercial boards could not have been used to realise the hardware architecture that was adopted, due to their hard wired link connections.

Interactive Image Processing Facility. To facilitate the realisation and invoking of algorithms, an interactive image processing system was designed and implemented using the tree architecture devised in the research. This software framework facilitated the investigation into the operation of realised algorithms, and their performance on the tree network. Single operations or streams of such operations could be executed on specifically created test images or dynamically captured scenes.

14.4 The Hardware and Software Architecture

The Hardware Architecture. The hardware architecture adopted for this research was an inverted order three tree, or ternary tree. This was chosen because of the inherent shortest distance from the root node to any other node, the high fan-out capability, and the fact that the tree is highly extendible. Communications are more straightforward, being either up or down the tree, and there is a degree of fault tolerance associated with the structure.

The Software Architecture. A software architecture was devised and implemented on a ternary tree network of transputers. The interaction of SUPPLY and DEMAND software processes allowed many requirements associated with message passing multi-processor systems to be satisfied. The software could be reconfigured to execute on a tree with a different number of processors by altering a parameter in the automatic configuration program.

14.5 The Tree Architecture Performance

Low Level Transformations. The results obtained from the implementation of low level transformations were far from linear. Due to the image sectioning required, and the inclusion of neighbouring pixel values where required, additional overhead data had to be distributed. Table 14.1 shows the overheads associated with the image sections used.

Image Section Size	Number of Sections	Overhead Percentage
16 by 32	32	19.5
16 by 16	64	26.6
8 by 16	128	40.6

Table 14.1. Overhead Pixels Associated with Image Sections.

Results for the most simple, least computationally intensive transformation, that of Negate, indicated performance increases shown in table 14.2.

Number of Transputers	Performance Increase Image Section Size 16 by 32	16 by 16	8 by 16
1	1	1	1
4	3.1	3.3	3.4
10	3.6	5.3	5.2
20	3.2	6.2	5.3
32	3.0	5.8	5.1

Table 14.2. Performance Increases for Negate.

Number of Transputers	Maximum Speed Up Obtained	Efficiency
4	3.4	85%
10	5.3	53%
20	6.2	31%
32	5.8	18.1%

Table 14.3. Performance Efficiencies for Negate.

From table 14.3, it can be seen that the highest performance increase compared to the parallel implementation executing on one transputer was 6.2, using 20 transputers. This represents an efficiency of 31%. The highest efficiency shown is 85%, using 4 transputers. The reason for the fall off in the speed up figures is because of the extra layers used in the tree network.

The most computationally intensive low level transformation implemented was that of Laplacian Convolution. Comparative speed increases for this are shown in table 14.4.

Number of Transputers	Performance Increase Image Section Size 16 by 32	16 by 16	8 by 16
1	1	1	1
4	3.6	3.8	3.7
10	7.5	8.9	8.3
20	8.7	13.1	13.0
32	9.8	15.9	12.7

Table 14.4. Performance Increases for Laplacian.

Number of Transputers	Maximum Speed Up Obtained	Efficiency
4	3.8	95%
10	8.9	89%
20	13.1	65.5%
32	15.9	49.7%

Table 14.5. Performance Efficiencies for Laplacian.

Referring to table 14.5, the highest performance increase obtained, compared to using one transputer, was 15.9 using 32 transputers, an efficiency of 49.7%. Using 20 processors yielded a speed up of 13.1 with an efficiency of 65.5%. The use of four transputers had a speed up of 3.8 with an efficiency of 95%.

Boundary Formation in Parallel. The Boundary of an object was chain coded by the multi-transputer network in

parallel. Each DEMAND process received sections of a grey level image. A local Threshold was applied to the section, then any edges present were coded into Partial Boundary chains, from portions of the binary object. Partial Boundary chains were joined to other Partial Boundary chains using a multi-stage overlapped distributed joining up technique. A summary of the results obtained is shown in table 14.6.

Number of Transputers	Performance Increase Image Section Size 16 by 32	16 by 16	8 by 16
1	1	1	1
4	3.7	3.8	3.65
10	9.2	9.3	9
20	14.8	14.4	13.8
32	15.3	20	16.2

Table 14.6. Performance Increases for Parallel Chain Code Generation.

Referring to tables 14.6 and 14.7, it can be seen that the highest speed up obtained was 20 using 32 transputers, and an image section of 16 by 16, resulting in an efficiency of 62.5%.

The reason that the use of 16 by 32 image sections yielded a speed up less than that of the 16 by 16 sections is because of the parallelism granularity that this gives. Using table 14.1, it can be seen that a 16 by 32 image section gives 32 sections for the image. When using 32

transputers, this is clearly one image section per DEMAND process. This gives too coarse a granularity to allow the algorithm to be divided efficiently between the available processors, and consequently the performance is degraded.

Number of Transputers	Maximum Speed Up Obtained	Efficiency
4	3.8	95%
10	9.3	93%
20	14.8	74%
32	20	62.5%

Table 14.7. Performance Efficiencies for Parallel Chain Code Generation.

It can be seen from table 14.6 that the performance using 8 by 16 image sections is also worse than that obtained using 16 by 16 sections. This is due to the granularity being too fine, as there are 128 image sections. It was found that approximately 100 Partial Boundary Chains were generated from the first stage of the algorithm, using a 8 by 16 image section (and test figure BLOB). This involved four distributed join up stages.

Convex Hull Formation in Parallel. Images were divided into sections, and distributed to the DEMAND processes, as with the Boundary Chain coding. Each DEMAND process applied a local Threshold operation to the grey level data, and produced Partial Boundary chains and their Partial Convex Hull chains from any binary object edges present in that data section.

Number of Transputers	Performance Increase Image Section Size 16 by 32	16 by 16	8 by 16
1	1	1	1
4	3.6	3.6	3.6
10	9.1	9.1	8.6
20	16.4	18	15.7
32	25.2	27.2	22.7

Table 14.8. Performance Increases for Parallel Convex Hull Formation.

Number of Transputers	Maximum Speed Up Obtained	Efficiency
4	3.6	90%
10	9.1	91%
20	18	90%
32	27.2	85%

Table 14.9. Performance Efficiencies for Parallel Convex Hull Formation.

Referring to tables 14.8 and 14.9, it can be seen that the maximum speed up obtained was 27.2 with 32 transputers (and a 16 by 16 image section), and the efficiency was 85%. The speed ups were found to be 87%, 90% and 80% of linear using 16 by 32, 16 by 16 and 8 by 16 data sections respectively, with up to 13 transputers. These figures were reduced to 71%, 83% and 57% respectively using 16 to 32 transputers.

The section size of 16 by 16 allowed a finer granularity to be used, enabling a more efficient algorithm division between a higher number of processors. Using an 8 by 16 section, however, resulted in too many small Partial Convex Hulls being generated from the initial stage, and required five stages in all, to build up the overall Convex Hull. This explains why the speed up figures are lower for the 8 by 16 section than for the 16 by 16 section.

Feature Extraction. Features were extracted from the complete boundary and Convex Hull chains, involving Area and Perimeter parameters, number and type of the convex hull deficiencies, and CHD area ratios.

A Feature Vector was constructed using Shape Factors and the other features extracted. An object was thus reduced to a sequence of numerical values.

Scene Interpretation. Using the feature vector, a Nearest Neighbour Classification was devised. Each DEMAND process would compute the squared Euclidean distance from a feature vector being classified and each of the pre-stored Feature Vectors from a database.

Summary. The tree architecture has been shown to be not particularly efficient for executing low level transformations, due to the data intensive and low computational requirements of transformations. The highest speed up obtained with a low level transformation, that of a Laplacian Convolution, was 15.9 times faster using 32 transputers than it was using one transputer, which was an efficiency of 49.7%. The lowest speed up, that of Negate, was 6.2 times faster using 20 transputers than for one transputer, an efficiency of 31%.

The tree architecture shows great potential for the implementation of higher order, data dependent image processing algorithms. The implementation of the Convex Hull had a speed up of 27.2 using 32 transputers compared to one device, an efficiency of 85%. The speed ups obtained were found to be 80% - 90% of linear using up to 16 transputers, and 57% - 83% using up to 32 transputers.

14.6 Extensions to the Work Performed

Digital Signal Processor Device. If a Digital Signal Processor (DSP) device was incorporated into the image data path, after the digitiser (figure 13.1), then it would be possible to perform the required low level transformations at high speed. Ideally only chain coded edges would be passed to the multi-transputer network. This would enable the tree network to be used solely for the feature extraction and shape analysis for which it is better suited.

The results then obtained would be far more linear, allowing greater speed ups to be obtained, and therefore faster complex algorithm execution.

Higher Order Trees. If there were more links per transputer, either in future processors (as proposed in Chapter 1), or by memory mapping link adapters into the memory space (as discussed in Chapter 2), then higher order tree architectures could be used.

Higher order trees would then have greatly enhanced benefits over the ternary tree. As can be seen in the graphs, the performance of low level transformations falls

off after 4 transputers. This is due to the increase in the depth of the tree. If this was extended, then the performance would continue to be near linear for more processors.

14.7 Concluding Remarks

The hardware and software architectures proposed and investigated in this work have been used with low level transformations, and higher level Boundary and Convex Hull formation. A preliminary implementation of a Feature Vector construction has been performed, and its uses to enable Learn and Recognise tasks has been studied.

The tree architecture has been shown to be particularly effective for the complex functions of Boundary Chain Coding, Convex Hull formation, Feature Vector calculation, and Object Classification. As discussed in Chapter 9, and illustrated in figure 9.1, a geometric mapping of these non determinate operations onto an array cannot achieve linear or near linear performance speed ups.

For the test figure, BLOB, (using 16 by 32 sections) it was found that 22% of the sections had no object boundary, 15.5% had very little of the boundary, and the remaining 62.5% had unequally sized boundary lengths. Clearly, a geometric array mapping would not be as efficient in these cases, and could achieve a theoretical maximum efficiency of 62.5%, although with the unequally sized boundary lengths in the sections, this figure would be substantially reduced. The software and hardware architectures proposed in this work achieved a maximum efficiency of 62.5% for

Boundary Chain Coding, and 85% for Convex Hull formation, using 32 transputers.

The use of a front end Digital Signal Processor device, together with higher order tree networks have been proposed, to make the overall system more efficient for these tasks.

Appendix A

List of Publications and Papers Presented

Occam Programming on an Apple Microcomputer - Research Report No. 85/IB/1, Gwent College of Higher Education, December 1985. *

The Transputer and Occam applied to Vision Systems - Seminar, Gwent College of Higher Education - 2 May 1986. *

Technical Reference Manual for Transputer Development System Z80 Interface, Research Report No. 86/IB/1, Gwent College of Higher Education, May 1986. *

Interfacing the Transputer, Research Report No. 86/IB/2, Gwent College of Higher Education, July 1986. *

Designing with the Transputer, Research Report No. 86/IB/3, Gwent College of Higher Education, October 1986. *

An Image Processing System with Optimising Performance, Occam User Group Sixth Technical Conference, Surrey University, Guildford, U.K., 15 April 1987. *

Parallel Processing Language - occam 2, Seminar, Gwent College of Higher Education, 8 May 1987. *

Image Processing with Transputers, Seminar, IBM (U.K.) Scientific Centre, Winchester, 19 May 1987. *

Shape and Pattern Recognition Using Transputers, Seminar, Gwent College of Higher Education, 19 June 1987. *

Submission to CNAA for Transfer of Registration from M.Phil to Ph.D, Research Report No. 87/IB/1, Gwent College of Higher Education, June 1987.

Concurrent Image Processing using occam and the Transputer, Research Report No. 87/IB/2, Gwent College of Higher Education, November 1987.

Multi-transputer based Parallel Implementation of Feature Extraction for Object Recognition, Occam User Group Eighth Technical Conference, Sheffield City Polytechnic, Sheffield, U.K., 29 March 1988. *

---

* With Dr. D.W. Downing (Ph.D Supervisor)

Object Recognition using Transputers, I.E.E. Younger Members' Short Papers Evening, UWIST, Cardiff, South Wales, U.K., 4 May 1988.

Image Feature Extraction using Transputers, I.E.E. Younger Members Technical Papers Evening, Swansea University, South Wales, U.K., 5 May 1988.

An Interactive Image Processing System based on a Transputer Tree Network, I.E.E. Colloquium on Transputers for Image Processing Applications, I.E.E., London, February 1989.

Appendix B

Papers Published

1. An Image Processing System with Optimising Performance, D.W. Downing and I.B. Bennett, Occam User Group Sixth Technical Conference, Surrey University, Guildford, U.K., 15 April 1987.

Abstract included of Paper Presented.

2. Multi-transputer based Parallel Implementation of Feature Extraction for Object Recognition, D.W. Downing and I.B. Bennett, Occam User Group Eighth Technical Conference, Sheffield City Polytechnic, Sheffield, U.K., 29 March 1988.

Paper published and presented included.

3. Object Recognition using Transputers, I.B. Bennett, I.E.E. Younger Members' Short Papers Evening, UWIST, Cardiff, South Wales, U.K., 4 May 1988.

Abstract included of Paper Presented.

4. Image Feature Extraction using Transputers, I.B. Bennett, I.E.E. Younger Members Technical Papers Evening, Swansea University, South Wales, U.K., 5 May 1988.

Abstract included of Paper Presented.

5. An Interactive Image Processing System based on a Transputer Tree Network, D.W. Downing and I.B. Bennett, I.E.E. Colloquium on Transputers for Image Processing Applications, I.E.E., London, February 1989.

Paper published and presented included.

1. An Image Processing System with Optimising Performance, D.W. Downing and I.B. Bennett, Occam User Group Sixth Technical Conference, Surrey University, Guildford, U.K., 15 April 1987.

Abstract included of Paper Presented.

An Image Processing System With Optimising Performance

D.W. Downing and I.B. Bennett

Abstract of a paper presented to The Sixth Occam User Group Technical Meeting, Surrey University, Guildford, April 1987

This paper describes a complete image processing system based exclusively on a network of transputers. Images from a 128 * 128 pixel, 256 grey level video camera / frame store are captured using link adapters. Concurrent processing takes place on a varied topology of transputers, and the processed images are displayed via link adapters on a video monitor.

More than 80 image processing algorithms have been implemented, ranging from Thresholding and Histogram Adaptation, to Edge Detection and Convex Hull formation. The processing gives a considerable speed improvement on many commercially available systems.

Various network configurations have been investigated. This paper illustrates a trinary tree structured network of 16 transputers in which self optimisation is implemented. The algorithm is robust and also adjusts itself to a variable number of nodes.

Experience in board design and construction using transputers and Link Adapters will also be covered.

A comparison of execution times and practical aspects of implementing such algorithms in occam and on transputer hardware will be discussed.

The system will be available for demonstration.

2. Multi-transputer based Parallel Implementation of Feature Extraction for Object Recognition, D.W. Downing and I.B. Bennett, Occam User Group Eighth Technical Conference, Sheffield City Polytechnic, Sheffield, U.K., 29 March 1988.

Paper published and presented included.

MULTI-TRANSPUTER BASED PARALLEL IMPLEMENTATION OF FEATURE EXTRACTION FOR OBJECT RECOGNITION

DR D.W. DOWNING Reader
I.B. BENNETT Research Assistant

Gwent College of Higher Education
Faculty of Technology
Allt-Yr-Yn Avenue,
Newport, Gwent, NP9 5XA
Correspondence may be addressed to Dr. D.W. Downing.

ABSTRACT

The main emphasis of this paper is on the parallel implementation of high level feature extraction for Object Recognition. Orientation independent topological image features such as area, perimeter, convex hull parameters and deficiencies, and shape compactness ratios are considered.
The authors use a ternary tree configuration of transputers. The software executing on the system is flexible, modular, and self adapting, allowing the number of processors used in the target system to be readily altered.
Identifying features are computed directly from the chain codes of object edges, to allow feature vectors to be constructed. This enables LEARN and RECOGNISE procedures for object recognition to be developed.

1.0 INTRODUCTION

This paper describes the parallel implementation of feature extraction in an interactive image processing system designed for automated visual inspection.
Object recognition is a highly computationally intensive task, requiring different levels of complexity in image processing operations, and varying data formats [1, 2, 3]. The concurrent processing capability and flexibility of transputers makes them ideal for this function.
Two-dimensional images of 128 * 128, 256 grey level pixels are captured, processed in parallel to extract features, and displayed in an interactive mode. In this way procedures best suited to an industrial application can be grouped as macros and used in streamlined mode to produce efficient parallel utilisation of transputer networks. The software can easily be extended to work with higher resolution images.
The system hardware consists of a PC-compatible based Transputer Development System, interfaced with link adaptors [4] to a CCD camera and digitiser, and also a display monitor. The interactive image processing system, using in-house designed and built quad-transputer boards [5], is shown in Figure 1.
A total of more than 90 image processing operations have been realised and validated on a single transputer system using occam 2. Most of these have subsequently been implemented on the multi-transputer network, using up to 9 transputers.

A fast front-end Digital Signal Processor (D.S.P.) device [6,7,8], could be used to implement the lower level image transformations. The D.S.P. device could also generate edges or chain coded edges directly from an image, passing these to the multi-transputer system. This approach was not adopted as it was the aim of the research to investigate the use of transputers for image processing operations of all levels of complexity.
The system design presented here enables simple replication of data distribution and processing on an arbitrary number of transputers in a tree structured network.
Emphasis is, however, placed on the methods that have been applied to parallel partitioning of the higher level procedures for image and scene analysis in transputer networks.
The system manipulates, enhances and thresholds images, using a variety of parallel algorithms. Object edges are coded to reduce the amount of data required to represent images. Chain codes of objects, their convex hulls and deficiencies are generated concurrently throughout the multi-transputer system.

2.0 IMAGE ANALYSIS OVERVIEW

There are essentially three levels of image processing, categorised by the following:

a) Image Enhancement
b) Feature Extraction
c) Scene Interpretation

Operations of all three levels are necessary in an efficient interactive system in order to select an optimum processing stream for object classification and recognition.
a) A set of over 60 image enhancement algorithms, which include spatial filtering, histogram equalisation, and edge detection, have been completed. These form a basic set of image conditioning procedures from which processing macros are easily constructed.
Although SIMD machines can perform these lower level image processing transformations at high speed, they are not as well suited as MIMD transputer networks to perform the more complex, higher order operations. The multi-transputer system also provides the additional flexibility required in interactive systems.
b) Spatial domain techniques for feature extraction include chain coded image data, convex hull deficiencies, and shape descriptors. The strategies for parallel implementation on tree networks will be described in Sections 3 and 4.
These higher level functions under development, are presently being executed and tested on Inmos Item 400 and Meiko systems under the support of The SERC Transputer Initiative at Rutherford Appleton Laboratory.
c) A simple visual inspection technique for industrial production processes is the subtraction of test images from ideal images, to reveal differences which are assumed to be faults. Such simple techniques rarely give entirely satisfactory results because of the variability in production, illumination, and image

noise. The difference image, therefore, requires some process of interpretation, which is often based on statistical knowledge and a database of information.
A tree network with its efficient hierarchical structure [9], is well suited both to object feature extraction, and scene interpretation. Widely differing algorithms can run concurrently, on different processors, and large databases can be distributed and searched in parallel.

3.0 SYSTEM OVERVIEW

A parallel hardware configuration and software architecture suitable for the task being investigated is shown in Figure 2. The network is a ternary tree structure in which all processors and processes, except for the process Supply in the root node, are replicated identically throughout the network. This feature adds consistency and expandability to the system [10]. All processors in the ternary tree network have an identical Demand process.
In outline, the Demand process consists of a full set of image processing procedures for enhancement, feature extraction and interpretation, which are executed by a sub-process Compute. Two data packet routing and distribution processes, Up and Down, execute in parallel with Compute. There is also a buffer process, for local temporary data packet storage.
Each data packet includes a data identifier tag, data format and size information, the operation to be performed on the data, and any operation specific parameters required.
The root processor has a Supply process executing in parallel with the Demand process, to initialise the network and supervise the data partitioning and distribution throughout the tree.
The work described deals specifically with feature extraction using chain code techniques [11]. Chain codes of an image are obtained by a sequence of thresholding, edge detection, and directionally coding step increments along the boundaries of all distinct objects in the image. A very efficient descriptor of objects is produced and the resulting chain code can be used to obtain further characteristic features, described in more detail in Section 4.
The potential of the interactive system is being investigated for automatic visual quality inspection of transputer chips in production at Inmos, Newport.
Generally, in the semiconductor industry today, wafers consisting of a number of dice are inspected manually by skilled operators for various types of production faults. Faults such as contamination, resist under exposure or track bridging are either random or repeated and inspection has to be performed on a sampling basis. At present the frequency and size of the faults are estimated and recorded manually.
Inspection requires several high level image processing functions in order to differentiate features and classify faults, and to give a quantitative measure of quality. Transputer networks offer the potential for real time processing.

4.0 PARALLEL FEATURE EXTRACTION ALGORITHMS

4.1 Chain codes

The chain code of an object's boundary is generated throughout the multi-transputer system in parallel. Each Demand process is assigned a variable sized rectangular section of the image. Any edges present in that image section are coded, chains are joined locally where possible, then passed to the root Supply process. This process invokes a joining algorithm within the Demand processes, which attempts to link non-closed chains, generated from adjacent image sections.
Chains can be linked to other chains in one of four ways, and may require reversing (see Figure 3). The boundary chain for an object is thus complete when a closed chain has been obtained. Non-closed or short boundary chains representing noise or unwanted non-object related edges are discarded.

4.2 Convex Hull

The Convex Hull (C.H.) of an object is calculated from the chain coded boundaries [12]. This algorithm has the advantages of utilising the low data volume inherent in chains, and avoiding any significant mathematics, thus yielding an efficient realisation.
When a chain representing an edge has been produced locally in a Demand process, the associated partial C.H. chain is computed. As boundary chains are joined to other chains, a new partial C.H. chain is computed from the original C.H. chains as shown in Figure 4.
If a boundary chain is found to be closed, then the complete C.H. is produced from the partial C.H. chains. This computation is greatly simplified by the use of the partial C.H. chains.

4.3 Convex Hull Deficiencies (C.H.D.s)

The chains representing bay C.H.D.s are computed from the object boundary and C.H. chains (Figure 5). Lake C.H.D.s are treated as separate objects, as they do not have any interaction with the boundary. These lake boundary chains are thus computed concurrently with object boundary chains. The system could easily be extended to produce a shape describing concavity tree.

4.4 Areas and perimeter lengths

Areas and perimeter lengths are directly computed from the chain codes. The system described distributes closed chains of object boundaries, object C.H.s, and C.H.D.s, for concurrent computation of areas and perimeters. These features of a closed chain are typically calculated by different processors.
Where there are few closed chains obtained from an image, the system has the flexibility for load distribution to allow computation of partial areas and perimeter lengths from parts of chains. Global figures for each chain are then obtained by the root process. This allows the utilisation of the multi-processor

configuration to be kept high.

4.5 Feature vector

The feature vector being used by the system at the time of writing (January 1988) has 5 component attributes. These comprise the shape factor of the object itself (F_o), the number and type of the C.H.D.s (N_l and N_b), their shape factors (F_i) and the ratios of their areas to the object (R_i),

where N_l is the number of lake C.H.D.s
N_b is the number of bay C.H.D.s
$i = 1, 2, 3 \ldots (N_l + N_b)$

The feature vector components are calculated in parallel, with the root process controlling and distributing the relevant parameters and data for the calculations to be performed.

4.6 Learn and Recognise

The feature vector generated can be added to a table, or database, which is at present stored on disk within the host system. This comprises the LEARN process.
RECOGNISE is achieved by performing a nearest-neighbour cluster analysis utilising a squared Euclidean distance measure [13] within each Demand process, using the feature vector obtained above, and the pre-distributed feature database. The classification method implemented involves computational complexity proportional to N * M, using an N-dimensional object describing mechanism, and M pre-learnt objects.

4.7 Performance and timings

A test figure, BLOB is shown in Figure 6a. The parameters of BLOB are given below. The convex hull operation is demonstrated in Figure 6b, and the 7 bay C.H.D.s are outlined in Figure 7a. Figure 7b clarifies the process adopted for the parallel generation of the boundary and C.H. chain codes. In the example, image sections of 32 * 16 are shown. A selection of timings of appropriate image processing operations on BLOB are given in table 1, for one and 6 transputers.

BLOB parameters

Image size	128 * 128
Area	4495 pixels
Perimeter length	714 pixel lengths
Shape factor F_o	113
No. lakes N_l	0
No. Bays N_b	7
C.H.D. shape factors F_i	29, 38, 24, 16, 23, 20, 15
C.H.D. Area ratios R_i	3, 5, 9, 9, 18, 27, 36

Feature vector for BLOB
113, 0, 7, 29, 38, 24, 16, 23, 20, 15, 3, 5, 9, 9, 18, 27, 36

Operation	Timings / mS	
No. of transputers	1	6
Negate	181	49
Laplacian Convolution (3*3 mask)	637	131
Binary edge	314	74
Expand	501	107
Shrink	314	74
Chain code boundary sections	389	66
Generate Convex Hull	257	*
Generation of C.H.D.s	238	*
Compute feature vector	50	*

Table 1

* Further results will be reported at the conference if they are available.

5.0 Conclusions

A multi-transputer based interactive image processing system has been described. The software and hardware structures implemented provided an extremely versatile and flexible working environment, allowing the degree of tree population to be readily altered. The authors found the Analyse and Debug facilities of the TDS useful in developing and debugging the complicated concurrent software. While the system can perform a wide range of image processing algorithms, it is best suited to higher level, more computationally intensive tasks.
The parallel implementation of the extraction of object features from an image has been discussed. This allowed simple forms of LEARN and RECOGNISE functions to be achieved.

6.0 Acknowledgements

The authors would like to thank Inmos staff, particularly Tony Gore, Stephen Brain, and Michael Poole for their interest, and help with TDS support. Use of Inmos Item 400 and Meiko systems provided by The SERC Transputer Initiative at Rutherford Appleton Laboratory is gratefully acknowledged.

References

[1] Digital Image Processing. W. Pratt, Wiley, 1978

[2] Automated Visual Inspection ed. B.G. Batchelor, D.A. Hill & D.C. Hodgson, IFS Publications Ltd. 1985

[3] Digital Picture Processing A. Rosenfeld and A.C. Kak 2nd Ed. New York Academic Press 1982

[4] Interfacing the transputer. I.B. Bennett and D.W. Downing Gwent College of Higher Education, Research Report No. 86/IB/2 July 1986

[5] Designing with the transputer. I.B. Bennett and D.W. Downing Gwent College of Higher Education, Research Report No. 86/IB/3 October 1986

[6] Software Tools for the NCR GAPP Processor. R.Raghavan Software Engineering for VLSI Parallel Processors IEE Digest No 1986/102, Oct. 1986

[7] A100 Technical Application notes. Inmos Ltd. 1986

[8] Developments in Image Processing for Industrial Inspection B.G. Batchelor et al SPIE Vol. 730, Automated Inspection and Measurement 1986

[9] Transputer arrays for Image Processing. C.J. Elliott IEE Colloquium on Image Processing for Industrial Inspection Digest No. 48, Part 6, pp1-5 1986

[10] An image processing system with optimising performance D.W. Downing and I.B. Bennett Occam user group sixth technical conference Surrey University, Guildford 15 April 1987

[11] Computer Processing of Line-Drawing Images H. Freeman Computing Surveys Vol. 6 No. 1 (March 1974)

[12] Convex Hull of a Chain Coded BLOB G.R. Wilson Private Correspondence

[13] Pattern Recognition, Ideas in Practice Ed. B.G. Batchelor Plenum Press 1978

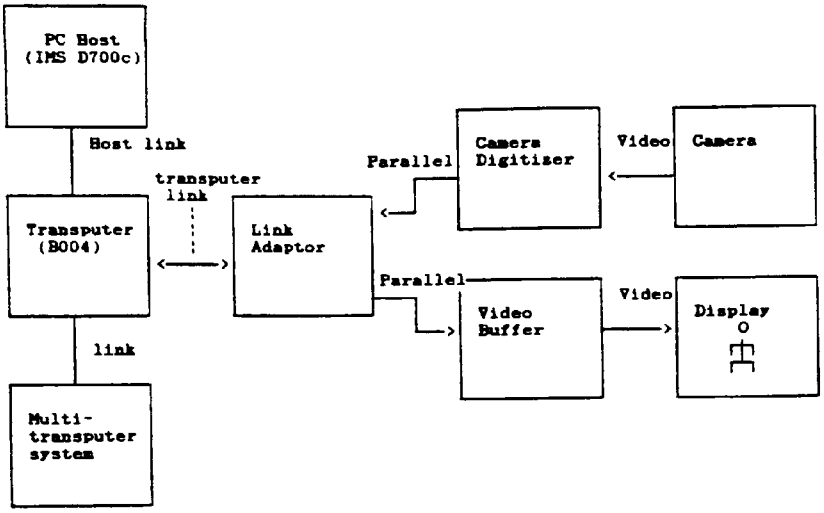


FIGURE 1. BASIC LABORATORY HARDWARE SETUP.

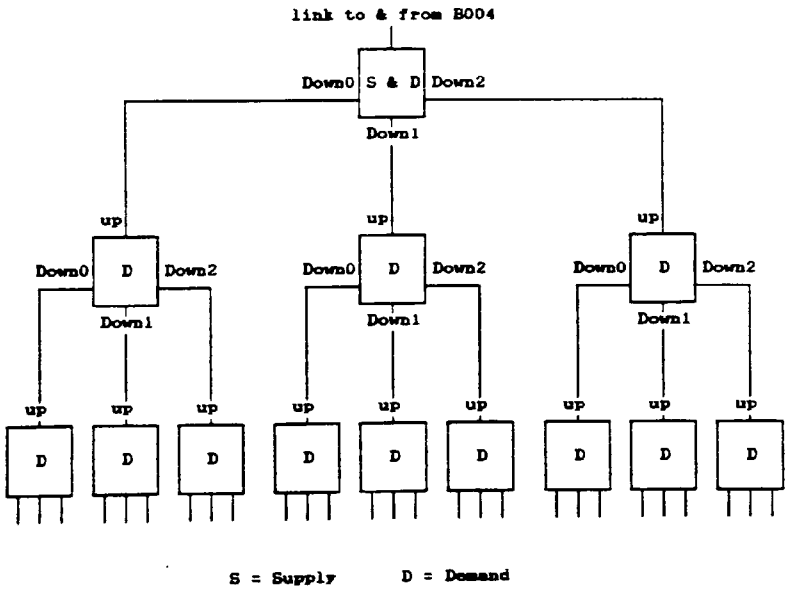


FIGURE 2. TREE CONFIGURATION USED, SHOWN POPULATED TO 3 LEVELS.

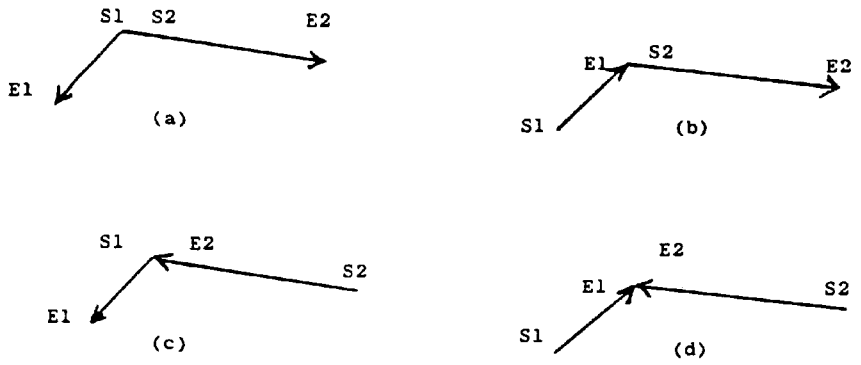


FIGURE 3. Chains can be joined in four ways; (a) Start to start (b) End to start (c) Start to end (d) End to end (NOTE b ≡ c).

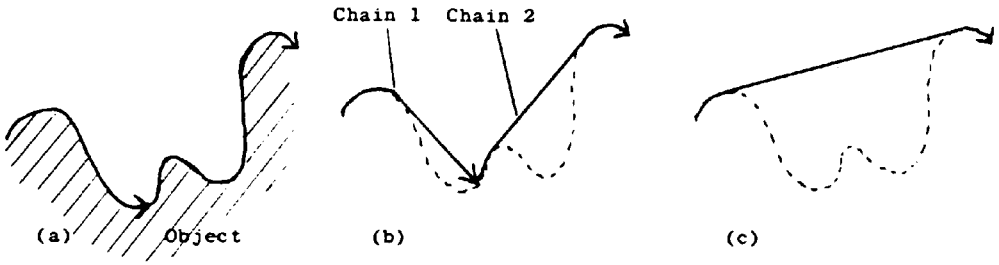


FIGURE 4 (a) Two joined object boundary chains
(b) Their respective partial C.H. chains
(c) The new partial C.H. chain produced

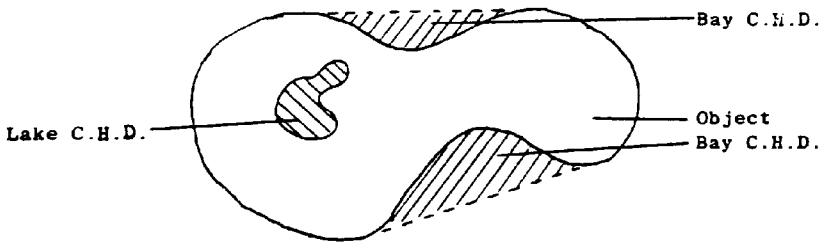


FIGURE 5. Chains representing bay C.H.D.s are produced from object boundary and C.H. chains. Chains for lake C.H.D.s are computed concurrently with the object boundaries.

(a)

(b)

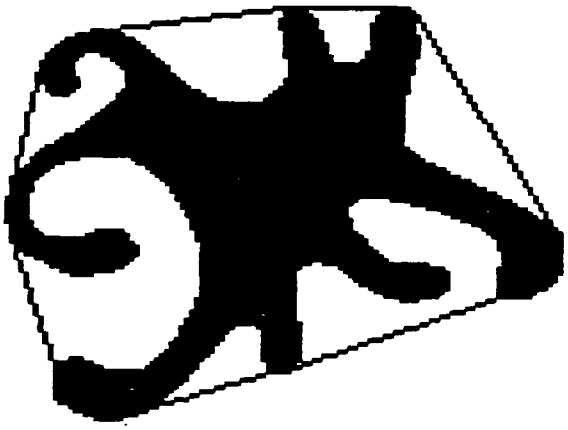
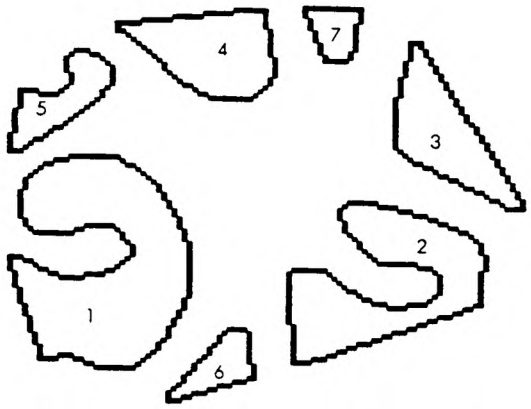


FIGURE 6. (a) Test object, BLOB.
(b) Convex hull operation, on BLOB.

(a)



(b)

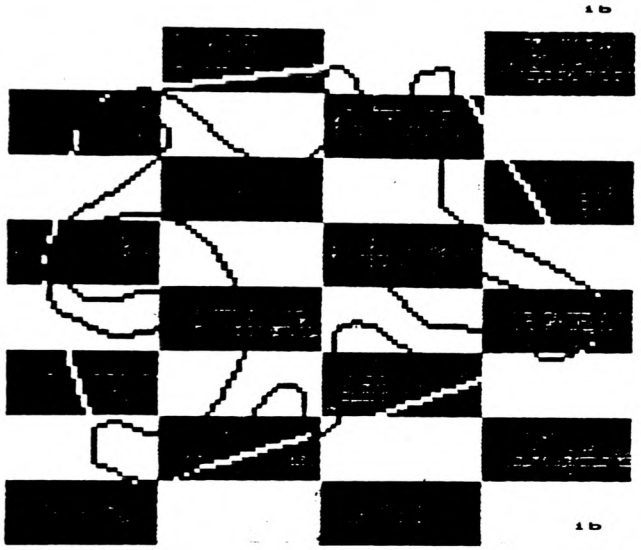


FIGURE 7. (a) Identification of the 7 Bay C.H.D.s. of BLOB.

(b) Example of parallel generation of chain codes, image segments shown are 32 x 16.

3. Object Recognition using Transputers, I.B. Bennett, I.E.E. Younger Members' Short Papers Evening, UWIST, Cardiff, South Wales, U.K., 4 May 1988.

Abstract included of Paper Presented.

Object Recognition using transputers

I.B. Bennett

Abstract of a paper presented to The I.E.E. Younger Members Technical Papers Evening, UWIST, Cardiff, May 1988.

The computationally intensive task of Object Recognition has many applications in Automated Industrial Inspection and Robot Vision. This paper describes the parallel implementation of image feature extraction to enable simple forms of Learn and Recognise procedures to be achieved. Techniques devised that are required for the effective utilisation of a multi-transputer system are presented.

The transputer is a very high performance 32-bit "computer on a chip", designed and made by a British company, Inmos. The 10 million instructions per second (Mips) device is ideally suited to computationally demanding and complex applications. The design philosophy of the transputer is fundamentally different from that of conventional microprocessors, allowing systems with many hundreds of devices to be successfully used.

The processor was designed to efficiently implement the parallel processing language occam. A new programming language was required to enable the correct specification and control of concurrent processes, allowing the true parallel processing capability of transputers to be realised.

Image processing operations can be categorised into three basic levels, image enhancement, feature extraction, and scene interpretation. An interactive image processing system has been designed, and executes on a multi-transputer configuration. Over 90 image enhancement, transformation and feature extraction algorithms have been realised and validated, most using up to 36 transputers. The hardware / software architecture devised is flexible, modular, and allows the number of transputer devices being used to be readily altered.

Orientation independent topological image features are computed directly from chain codes of object edges. A feature vector is then constructed involving convex hull parameters and deficiencies, areas, perimeters, and shape compactness ratios. The feature vector can then be added to a table (Learn), or used to perform a nearest-neighbour classification with the pre-distributed feature database (Recognise).

A selection of timings for appropriate image processing operations will be given.

4. Image Feature Extraction using Transputers, I.B. Bennett, I.E.E. Younger Members Technical Papers Evening, Swansea University, South Wales, U.K., 5 May 1988.

Abstract included of Paper Presented.

Image Feature Extraction using Transputers

I.B. Bennett

Abstract of a paper presented to The I.E.E. Younger Members Short Papers Evening, U.C. Swansea, May 1988.

Features that describe or identify objects can be used in many applications, including Automated Industrial Inspection and Robot Vision. Image feature extraction is a computationally intensive task, requiring the use of high performance programmable computing devices. This paper presents aspects and techniques required for the effective utilisation of a multi-transputer system for this task.

The transputer is a very high performance 32-bit "computer on a chip", designed and made by a British company, Inmos. The 10 million instructions per second (Mips) device is ideally suited to computationally demanding and complex applications. The design philosophy of the transputer is fundamentally different from that of conventional microprocessors, allowing systems with many hundreds of devices to be successfully used.

The processor was designed to efficiently implement the parallel processing language occam. A new programming language was required to enable the correct specification and control of concurrent processes, allowing the true parallel processing capability of transputers to be realised.

Over 90 image enhancement, transformation and feature extraction algorithms have been realised and validated, most using up to 36 transputers. The hardware / software architecture devised is flexible, modular, and allows the number of transputer devices being used to be readily altered. Workload distribution and balancing is achieved using dynamic work farming techniques.

Edges present in grey level images are chain coded, to reduce the data volume required. The convex hull of the object boundary is also generated concurrently throughout the multi-transputer system. Orientation independent topological features are computed directly from these boundary and convex hull chains. A feature vector is then constructed involving convex hull parameters and deficiencies, areas, perimeters, and shape compactness ratios. The feature vector can then be added to a table (Learn), or used to perform a nearest-neighbour classification with the pre-distributed feature database (Recognise).

5. An Interactive Image Processing System based on a Transputer Tree Network, D.W. Downing and I.B. Bennett, I.E.E. Colloquium on Transputers for Image Processing Applications, I.E.E., London, February 1989.

Paper published and presented included.

AN INTERACTIVE IMAGE PROCESSING SYSTEM BASED ON A TRANSPUTER TREE NETWORK

D W Downing and I B Bennett

INTRODUCTION

This paper describes the parallel implementation of an interactive image processing system based on a ternary tree configuration of transputers. The tree can be extended to utilise available processors.

Approximately 100 image processing algorithms have been implemented, involving low level transformations, feature extraction, and preliminary scene interpretation. Dynamic load balancing techniques have been utilised extensively in order to distribute the workload and data around the processors for these operations.

An application, such as object recognition is a highly computationally intensive task, requiring different levels of image processing operations using varying data formats (1, 2, 3). The parallel processing potential and flexibility of transputers makes them ideal for this function.

SYSTEM OUTLINE

The system hardware consists of the PC based, D700D, Transputer Development System, interfaced with link adaptors (4) to a CCD camera and digitiser, and also a display monitor. A hardware configuration using in-house designed and built quad-transputer (4xT414) boards (5) form the tree network. This network has been extended to 36 transputers by linking in to the Meiko and Inmos Item 400 facilities at the SERC/DTI Rutherford Appleton Laboratories Centre.

To enhance processing speeds a front-end Digital Signal Processor device could be used to directly generate edges or chain coded edges from an image, passing these to the multi-transputer system (6). This approach was not adopted as it was the aim of the research to implement image processing operations of all levels of complexity, and to retain a greater degree of flexibility required for interactive processing.

Images of 128 * 128, 256 grey level pixels are captured, processed in parallel to extract features, and displayed in an interactive mode. In this way procedures best suited to an industrial application can then be grouped as macros and used to produce efficient parallel utilisation of transputer networks. The software structure is flexible, modular, and self-adapting to the number of processor devices present. All processing is implemented in occam 2.

D W Downing and I B Bennett are at Gwent College of Higher Education, Newport, Gwent.

Some 100 image processing operations have been developed and validated on the transputer networks. The system design enables simple replication of data distribution and processing on an arbitrary number of transputers in a tree structured network.

SCOPE OF PROJECT

The aims of the project are to investigate the application of parallel and concurrent processing with transputer networks in the following areas of image processing.

(1) Image Enhancement
(2) Feature Extraction
(3) Scene Interpretation
(4) Practical Applications

(1) A set of image enhancement algorithms, which include spatial filtering, histogram equalisation, and edge detection, have been completed. These form a basic set of image conditioning procedures from which processing macros are easily constructed. It is proposed to extend these algorithms using frequency domain techniques so that more complex filtering can be implemented.

(2) Recent work has concentrated on parallel Feature Extraction. Spatial domain techniques for feature extraction include chain coded image data, convex hull deficiencies, and shape descriptors. The strategies for parallel implementation on tree networks will be described.

(3) Future work is proposed for Scene Interpretation on parallel systems, where it is envisaged that some advantages of tree structured networks in parallel database searches can be investigated.

(4) Applications are now being considered in the areas of robotic control, automated visual inspection. These require, however, optical and sensing equipment not at present available to the authors.

FEATURE EXTRACTION

Feature extraction is of fundamental importance in achieving practical and useful higher level, more intelligent functions.

Before features can be extracted from images, however, some low level operations must be applied. Typically background subtraction, ennancement, or filtering would be required. Thresholding would then yield a binary image, that can be used in subsequent edge detection and coding processes.

The authors [7] propose and use a ternary tree configuration, in order to improve the utilisation of the transputers when performing the more complex, higher order image processing operations. In substantial tasks, these constitute by far the majority of the computation time.

Initially, topological features such as area, perimeter, convex hull parameters and deficiencies are being extracted from 2-D image representation of 3-D objects. Fourier Descriptors will be implemented to investigate the effectiveness of more mathematical features.

The strategy used in recent research is to code the objects' boundaries, using a Freeman chain code [8], thus significantly reducing the amount of data.

The system at present uses 128 * 128 grey level images, which occupy 16 Kbytes of memory. The software is such that 256 * 256 or larger images could be readily used. Chain coding the edges reduces the data volume to the order of a few hundred bytes, which is more efficient for inter-transputer data communications purposes.

The chain code representing an object's boundary is generated throughout the multi-transputer system in parallel. Convex hulls are computed in a similar fashion. Chains representing the lake and bay convex hull deficiencies are then obtained directly from these two chains. Areas, perimeters and shape compactness factors are all then computed in parallel, from the chain codes.

An N-dimensional feature vector is thus created, representing the identifying attributes of the original object.

A matching process is proposed which performs a Nearest-Neighbour cluster analysis, utilising a squared Euclidean distance function (1, 3), in parallel on the multi-transputer configuration. This typically performs a "best match" classification out of the O objects coded in the database, in approximately 1/T time, using T transputers.

References

(1) Digital Image Processing W Pratt. Wiley 1978

(2) Automated Visual Inspection IFS Publications Ltd. 1985
ed. B G Batchelor, D A Hill & D C Hodgson

(3) Digital Picture Processing A Rosenfeld and A C Kak
2nd Ed. New York Academic Press 1982

(4) I B Bennett and D W Downing "Interfacing the transputer" July 1986
Gwent College of Higher Education

(5) I B Bennett and D W Downing "Designing with the transputer"
October 1986, Gwent College of Higher Education

(6) A100 Technical Application notes Inmos Ltd. 1986

(7) D W Downing and I B Bennett "An Image Processing System with Optimising Performance" Occam User Group Sixth Technical Conference
Surrey University, Guildford 15 April 1987

(8) H Freeman "Computer Processing of Line-Drawing Images"
Computing Surveys Vol 6 No. 1 (March 1974)

Appendix C

Image Processing Algorithms Implemented

Algorithm Description

Add two images.
Area computation.
Automatic Threshold.
Bit Complement.
Bit Set.
Chain Code Boundary Edge (in parallel).
Chessboard test figure.
Contrast Enhance.
Convex Hull in image space.
Convex Hull from a chain code.
Convex Hull (parallel implementation).
Copy image.
Count White Neighbours.
Count White Points.
Cursor mode.
Development test routine.
Digitise and Capture image.
Direction of Brightest Neighbour.
Divide images.
Double all intensities.
Exchange images.
Exclusive-OR between two images.
Expand White Regions.
Filled Rectangular Test Figure (box).
Freeman Chain Code.
General purpose Convolution.
Gradient.
Halve all Intensities.
Halve Axes.
Highlight Intensity Range.
Histogram Equalisation Enhancement.
Histogram Plot.
Horizontal Gradient.
Image Multiply by constant.
Image := temporary image.
Intensity Wedge along x axis.
Intensity Wedge along y axis.
Invert x and y Axes.
Invert x Axis.
Invert y Axis.
Keep Significant Bits.
Laplacian Filter.
Largest Neighbour.

List Operations Available.
Low Pass Filter.
Lower Intensity.
Maximum Pixels of two images.
Maximum x value in binary image.
Maximum y value in binary image.
Minimum Pixels of two images.
Minimum x value in binary image.
Minimum y value in binary image.
Multiply images.
Negate.
Perimeter Computation.
Percentage Threshold.
Point Remove.
Read image from disk.
Roberts Edge.
Rotate image (wraparound).
Row Maximum.
Shift image.
Shift Intensity.
Shrink White Regions.
Smallest Neighbour.
Sobel Edge.
Square Intensities.
Stop.
Subtract images.
Temporary image := image.
Threshold.
Timing control toggle.
Type string.
Upper Intensity.
Vector Plot.
Vertical Gradient.
Write image to disk.
Zeroise image.

Appendix D

Test Figure BLOB.

LIST OF REFERENCES

[1] An Enhanced Mesh Connected VLSI Architecture for Parallel Image Processing - V. K. P. Kumar, C. S. Raghavendra, I.E.E.E. Proceedings International Conference on Computer Vision and Pattern Recognition, San Francisco, CA, USA, June 1985

[2] Image Transform Coding: A Case Study Involving Real Time Signal Processing - G. E. Brebner, R. T. Ritchings, I.E.E. Proceedings Vol 135, Pt E, No. 1, January 1988

[3] Developments in Image Processing for Industrial Inspection - B. G. Batchelor et al, S.P.I.E. Vol. 730, Automated Inspection and Measurement 1986

[4] VLSI Convolvers for Computer Vision - A. Perez, Proceedings International Conference on VLSI and Computers, Hamburg May 1987

[5] M. J. B. Duff & T. Fountain - Cellular Logic Image Processing, Academic Press, London 1986

[6] Software Tools for the NCR GAPP Processor - R. Raghavan, Software Engineering for VLSI Parallel Processors, I.E.E. Digest No 1986/102, October 1986

[7] Transputer Arrays for Image Processing - C. J. Elliott, I.E.E. Colloquium on Image Processing for Industrial Inspection, Digest No. 48, Part 6, pp1-5, 1986

[8] Signal Processing with Transputer Arrays (TRAPS) - J. G. Harp et al, Computer Physics Communications, Vol 37 pp77-86, 1985

[9] (Ed.) B.G. Batchelor, D.A. Hill & D.C. Hodgson, Automated Visual Inspection, IFS Publications Ltd., 1985

[10] Trends in VLSI Microprocessor Design - D. Alpert et al, Proceedings I.E.E.E. Conference on VLSI and Computers, Hamburg, May 1987

[11] Next Generation Computer Systems - B. J. Procter, I.E.E.E. Proceedings on VLSI and Computers, Hamburg, May 1987

[12] The Influence of VLSI Technology on Computer Architecture - D. May, B.C.S. & Occam User Group Conference on Parallel Architectures for Artificial Intelligence, University College, London, Feb 1988

[13] Personal comments at Ninth Occam User Group Technical Conference - A. Hey, Southampton University, September 1988, U.K.

[14] Communicating Process Computers - D. May, R. Shepherd, Technical Note, Inmos Ltd., 1988

[15] R. W. Hockney C. R. Jesshope, Parallel Computers - Adam Hilger Ltd., Bristol 1981

[16] Parallel Processing with the DisPuter - C. P. Winder, Occam User Group Eighth Technical Conference, Sheffield City Polytechnic, Sheffield, March 1988

[17] Special Computer Architectures for Pattern Recognition and Image Processing, An Overview - K. S. Fu, Proceedings A.F.I.P.S., 1978 National Computing Conference, Vol 47, June 1978, pp1003-1013

[18] W. Pratt, Digital Image Processing, Wiley, 1978

[19] Some Computer Organisations and their Effectiveness - M. J. Flynn, I.E.E.E. Transactions on Computing, 948, Sept 1972

[20] CLIP4: A Large Scale Integrated Circuit Array Parallel Processor - M. J. B. Duff, International Joint Conference on Pattern Recognition, 4 pp 728-733 (1976)

[21] Bus Scheduling for a Multiple-Processor System - P. Markenscoff, I.E.E. Proceedings Vol 134 pt E No. 6 Nov 1987

[22] M. Ben-Ari, Principles of Concurrent Programming - Prentice Hall Int. 1982

[23] Essential Issues in Multi-Processor Systems - D. D. Gajski, J. K. Peir, I.E.E.E. Computer June 1985 Vol18 No.6

[24] Exploiting Concurrency - A Ray Tracing Example - J. Packer, Inmos Ltd. Technical note 7, 1987

[25] Transputer Reference Manual - Inmos Ltd., 1986

[26] Lies, Damned Lies, and Benchmarks - R. Shepherd, Inmos Ltd. Technical note, 1988

[27] (ed.) H.J. Mitchell, 32-bit Microprocessors, Collins 1986

[28] IMS T800 Technical Data Sheet, Inmos Ltd., 1987

[29] Occam 2 Language Definition - D. May, Inmos Ltd., Technical Publication 1987

[30 The Compiler Writer's Guide - Inmos Ltd., 1987

[31] Occam Programming on an Apple Microcomputer - I. B. Bennett and D.W. Downing, Research Report No. 85/IB/1, Gwent College of Higher Education, December 1985

[32] Real Time Processing of Large Volume Data from Photographic Plate Measurements - W. A. Cormack et al, Occam User Group Eighth Technical Conference, Sheffield City Polytechnic, Sheffield, March 1988

[33] Image Feature Extraction using Transputers - I. B. Bennett, I.E.E. Younger Members Technical Papers Evening - Swansea University, South Wales, U.K. 5 May 1988

[34] Pattern Recognition - A. Rosenfeld, J.S. Weszka (appeared in (Ed.) R. Bernstein, Digital Image Processing for Remote Sensing, USA, 1978)

[35] Object Recognition using Transputers - I. B. Bennett, I.E.E. Younger Members Short Papers Evening, UWIST, Cardiff, South Wales, U.K. 4 May 1988

[36] M. J. B. Duff and S. Levialdi - Languages and Architectures for Image Processing, Academic Press London 1981

[37] K. Hwang and F. A. Briggs, Computer Architecture and Parallel Processing - McGraw-Hill 1986

[38] The Gap Between Software Implementation and Hardware Realisation of Image Processing - H. Kazmierczak, Proceedings International Conference on VLSI and Computers, Hamburg, May 1987

[39] International Parallel Processing Projects: A Software Perspective - I.E.E.E. Software, Vol2 pt4, July 1985 pp65-80

[40] Alvey Programme Annual Report - Poster Supplement, I.E.E., B.C.S., and Alvey Directorate, 1987

[41] A Comparison of Hardware and Software Aspects of Recent Advances in Polyprocessor Architectures - N. S. Scott and P. Milligan, Occam User Group Sixth Technical Conference, Surrey University, April 1987

[42] Image Processing Architectures - V. Buzuloiu, Proceedings International Conference on VLSI and Computers, Hamburg May 1987

[43] A Pyramid Project using Integrated Technology - V. Cantoni et al, (Appeared in Integrated Technology for Parallel Image Processing, ed. S. Levialdi Academic Press 1985 London)

[44] Aspects of Multi-Processor Systems - A Brief Survey and New Concepts - W. H. Burkhardt, Proceedings International Conference on VLSI and Computers, Hamburg, May 1987

[45] Designing Parallel VLSI Architectures for Low Level Image Processing Applications - A. Krikelis, Proceedings International Conference on VLSI and Computers, Hamburg, May 1987

[46] Image Processing on the IBM PC - H. J. Myers and R. Bernstein, Proceedings I.E.E.E., Vol73 N6, June 1985

[47] A Modula-2 Based Image Processing Development System - C. M. Worrall and M. A. Browne, I.E.E. Software Engineering Journal, March 1986

[48] The Illinois Pattern Recognition Computer ILLIAC III - B. H. McCormick - I.E.E.E. Transactions, EC-12, 791 - 813

[49] Design Tools for High Speed Arithmetic Processors (Alvey Project) - University of Warwick

[50] Wafer-Scale Integration of Two Dimensional Processor Arrays (Alvey Project) - University of Oxford

[51] A Fault Tolerant Design Methodology for Wafer Scale Integration (Alvey Project) - Plessey Research (Caswell)

[52] Bit Level Systolic Arrays (Alvey Project) - Queen's University, Belfast

[53] A Low Cost Real Time Imaging and Processing System - H. F. Li et al, I.E.E. Software and Microsystems, V2 N5, Oct 1985

[54] RAPAC: A High Speed Image Processing System - A. C. Elphinstone et al, I.E.E. Proceedings V134 Pt E N1, Jan 1987

[55] ZR33481 Data Sheet, Zoran Corporation Inc., Santa Clara, CA, USA, 1985

[56] A100 and A110 Application notes - Inmos Ltd., 1986/8

[57] L642xx Family of Devices Data Sheet - L.S.I. Logic Ltd., 1987

[58] A Flexible Architecture for Image Processing - R. W. Hartenstein et al, Conference on Microprocessors and Microprogramming, Portsmouth, UK, September 1987

[59] ADSP-1010 16 * 16 Multiplier / Accumulator Data Sheet, Analog Devices Ltd., 1988

[60] A Real Time Image Compressor using a Modular Signal Processing System Employing occam and the Transputer - R. G. Bramley and D. J. Creasey, International Conference on Microprocessors and Microprogramming, Portsmouth UK, Sept 1987

[61] A High Speed Image Processing System using the TMS32010 - D. M. Holburn and I. D. Sommerville, I.E.E. Software and Microsystems, Vol 4, Nos. 5 and 6, October / December 1985

[62] Parallel Image Processing System based on the TMS32010 Digital Signal Processor - K. N. Ngan et al, I.E.E. Proceedings V134 Pt E N2, March 1987

[63] A Flexible Pipelined Architecture VLSI for Image Processing - T. Temma et al, Proceedings International Conference on VLSI and Computers, Hamburg, May 1987

[64] A Pipelined Processor for Low Level Vision - D. B. Gennery, B. Wilcox, I.E.E.E. Conference on Computer Vision for Pattern Recognition, San Francisco, CA, USA, June 1985

[65] High Speed Image Processing for Machine Vision - Ph.D Thesis, C. C. Bowman, University of Wales, 1986

[66] Multi-Processor Architectures for Machine Vision and Image Analysis - R. M. Lougheed et al, I.E.E.E. International Parallel Processing Conference, 1985

[67] Applying the transputer - P. Mattos, I.E.E. Electronics and Power, June 1987

[68] The transputer - P. Walker, BYTE, May 1985 pp219-235

[69] Transputer Application to Speech Recognition - J. Vaughan et al, Microprocessors and Microsystems, V11 Pt 7 September 1987, pp377-382

[70] Design of an Adaptable Pipeline based on Transputers - A. Basu - Proceedings International Conference on VLSI and Computers, Hamburg, May 1987

[71] The Manchester Prototype Data Flow Computer - J.R. Gurd et al, Communications of the A.C.M., Vol 28 No 1 Jan 1985, pp34 - 52

[72] An Architecture for Real Time Logic Programming in Process Control - D. Bosely and M. Woods, B.C.S. Conference on Parallel Architectures for Artificial Intelligence, University College, London, February 1988

[73] IMS C004 Cross Bar Switch Technical Data Sheet - Inmos Ltd., 1988

[74] A. Carling, Parallel Processing - the transputer and occam, Sigma Press, U.K. 1988

[75] Efficient High Speed Computing with the Distributed Array Processor - P. M. Flanders et al, (Appeared in High Speed Computer and Algorithm organisation, ed. D. J. Kuck et al, Academic press, New York 1977)

[76] Array Architecture - A. Lincoln, Systems International, November 1985

[77] Overview of the High Level Language for PICAP - B. Gudmundsson, (Appeared in Languages and Architectures for Image Processing, Ed. M. J. B. Duff and S. Levialdi, Academic Press, London 1981)

[78] The Illiac IV Computer - G. Barnes et al, I.E.E.E. Transactions on Computing, Vol C17, pp746-757, Aug 1968

[79] The Illiac IV system - W. J. Bouknight et al, Proceedings I.E.E.E., Vol 60, pp369-388, April 1972

[80] A Computer Oriented towards Spatial Problems - S.H. Unger, Proceedings I.R.E., 46, 1744 - 1750, 1958

[81] The Solomon Computer - D.L. Slotnick et al, Proceedings AFIPS, Autumn Computer Conference, pp87-107, 1962

[82] Parallel Processors for Digital Image Processing, M. J. B. Duff, (Appeared in Advances in Digital Image Processing, Ed. P. Stucki, Plenum Press, New York, 1979)

[83] Parallel Processing Pattern Recognition System UCPR1 - M. J. B. Duff et al, Nuclear Instrumentation and Methods, V52, pp284 - 288

[84] Cellular Logic and its Significance in Pattern Recognition - M. J. B. Duff, A.G.A.R.D. Proceedings Conference on Artificial Intelligence, No. 94, 25-1, 1971

[85] Further Developments - T. J. Fountain, (Appeared in Cellular Logic Image Processing - M.J.B. Duff and T. Fountain, Academic Press, London 1986)

[86] Man-Machine Interface Strategy for Image Processing - Alvey Directorate, 1987

[87] The Application of Cellular Logic to Image Processing - D. M. Watson - Ph.D Thesis, University of London, 1974

[88] PASM: A Partitionable SIMD / MIMD System for Image Processing and Pattern Recognition - H. J. Siegel et al, I.E.E.E. Transactions on Computers, Vol C-30 Dec 1981 pp 934-947

[89] An Intelligent Operating System for Executing Image Understanding Tasks on a Reconfigurable Parallel Architecture - E.J. Delp et al, I.E.E.E. International Conference on Computer Architecture for Pattern Analysis and Image Database Management, Miami Beach, Florida, Nov 1985

[90] SUPRENUM: The German Supercomputer Architecture, Rationale and Concepts - P.M. Behr et al, Proceedings I.E.E.E. International Conference on Parallel Processing, August 1986, Pennsylvania USA

[91] The Design of a Massively Parallel Processor - K. E. Batcher, I.E.E.E. Transactions on Computing, V29 pp 836 - 840, 1980

[92] Image Processing on the Reconfigurable Transputer Processor - J. G. Harp et al, Proceedings of the Occam User Group Seventh Technical Conference, Grenoble, Sept 1987

[93] A Medium Grained Parallel Computer for Image Processing - R. Cok, Proceedings of the Occam User Group Eighth Technical Conference, Sheffield City Polytechnic, Sheffield, March 1988

[94] Meiko Computing Surface Technical Handbook, Meiko Ltd., 1987

[95] Computer Vision for Robotics Using a Transputer Array - F. Fallside et al, I.E.E. Colloquium on Computer Vision for Robotics, London, January 1989, Digest No. 1989/9

[96] K. Hwang and A. F. Briggs, Chapter 7 of Computer Architecture and Parallel Processing, McGraw-Hill, 1986,

[97] The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture - G. F. Pfister et al, I.E.E.E. Parallel Processing Conference, 1985

[98] Transputer Implementation of the Radon Transform - G. Hall, I.E.E. Colloquium on Transputers for Image Processing Applications, London, February 1989, Digest No. 1989/22

[99] An Extension of the Processor Farm Using a Tree Architecture - S. A. Green, D. J. Paddon, Proceedings of the Ninth Occam User Group Technical Conference, Southampton University, IOS, 1988

[100] HAM - A Real Time Multi-Microprocessor Vision System - N. G. Bourbakis et al, I.E.E.E. Conference on Computer Vision and Pattern Recognition, Miami Beach Florida, June 1986

[101] A Transputer Architecture for Real Time Video Processing - A. N. Ham, I.E.R.E. Digital Processing of Signals in Communications, pp193-198 1985

[102] On The Performance of Tree Structured Machines for Window Based Image Processing - H. A. H. Ibrahim et al, I.E.E.E. Conference on Computer Vision and Pattern Recognition, Miami Beach Florida, June 1986

[103] The Pyramid Computer for Image Processing - R. Miller et al, I.E.E.E. Conference on Pattern Recognition, Montreal Canada, July 1984

[104] A Pyramid Implementation using a Reconfigurable Array of Processors - P. A. Sandon, I.E.E.E. Conference on Computer Architecture for Pattern Analysis and Image Database Management, Miami Beach, Florida, November 1985

[105] Sphinx, A Pyramidal Approach to Parallel Image Processing - A. Merigot et al, I.E.E.E. Computer Architecture for Pattern Analysis and Image Database Management, Miami Beach, Florida, November 1985

[106] A Pyramidal Architecture for Knowledge Based Sonar Image Interpretation - D. M. Lane et al, I.E.E. Colloquium on Transputers for Image Processing Applications, London, February 1989, Digest No. 1989/22

[107] Interfacing the Transputer - I. B. Bennett and D. W. Downing, Research Report No. 86/IB/2, Gwent College of Higher Education, July 1986

[108] RECBAR: A Reconfigurable Massively Parallel Processing Architecture - V. Balasubramanian and P. Banerjee, Proceedings of the I.E.E.E. Parallel Processing Conference, August 1986

[109] Signal Processing with occam and the Transputer - R. Taylor, I.E.E. Proceedings V131 Pt F N6, Oct 1984

[110] Caltech / JPL MkII Hypercube Concurrent Processor - J. Tuazon et al, Proceedings of the I.E.E.E. Parallel Processing Conference, 1985

[111] The Cosmic Cube - C.L. Seitz, Communications of the A.C.M., January 1985

[112] Viable Environments - R. Chamberlain and D. Moody, Systems International, January 1988 pp45-49

[113] The iPSC/2 - Product Feature, What's New In Computing, October 1987, p 58

[114] The Connection Machine - W. D. Hillis, Scientific American, Vol 256, No. 6, (June 1987), pp86-93.

[115] Autoview Reference Manual - British Robotics, 1985

[116] The AUTOVIEW Interactive Image Processing Facility - B. G. Batchelor et al, (Appeared in Digital Image Processing, (Ed.) N. B. Jones, Peregrinis for the I.E.E., 1982

[117] Series 150/151 Product Brief - High Performance Modular Image Processing Subsystem - Amplicon Electronics Ltd., Brighton, E. Sussex, 1988

[118] Display 3000 PC Image Processing Software, Data Sheet, Digital Imaging Systems Ltd., Bristol

[119] The MDP-4 Image Processor, Data Sheet - CDA Ltd., Massachusetts, USA

[120] The MicroMSP-4 Array Processor, Data Sheet - CDA Ltd., Massachusetts, USA

[121] The QT-Series, MicroVax Based Parallel Processing, Data Sheet - Caplin Cybernetics Corporation, London.

[122] The Maxvideo Range of Image Processing Boards, Data Sheet - Datacube Corporation, Massachusetts, USA.

[123] Image-Pro, a PC-Based Image Processing Software System, Data Sheet - KGB Micros Ltd., Berks, U.K.

[124] Image Analysis System, Data Sheet - Loughborough Sound Images Ltd., Leicestershire, U.K.

[125] A Framestore and Associated Software for the BBC Microcomputer, Data Sheet - Data Harvest Group Ltd., Bedfordshire, U.K.

[126] Visionix - A High Resolution Industrial Imaging System, Data Sheet - Digital Design, France.

[127] Developing Transputer Application Circuits - S. Ghee, Electronic Product Design, Aug 1986

[128] Applying the Transputer - S. Brain, Electronic Product Design, Jan 1984 pp43 - 48

[129] IMS B003 - Design of a Multi-transputer Board - A. Vadher, P. Walker, INMOS application note 2, 1986

[130] Designing with the Transputer - I. B. Bennett and D. W. Downing, Research Report No. 86/IB/3, Gwent College of Higher Education, October 1986

[131] A Characterisation of Algorithms for Parallel Computation - N. P. Holt, I.E.E. Colloquium on Transputer Applications, Digest No. 91, 1986

[132] Submission to CNAA for Ph.D / M.Phil Registration - D. W. Downing and I. B. Bennett, Gwent College of Higher Education, 1985

[133] Transputer Link Adapter Reference Manual - Inmos Ltd., 1985

[134] An Interactive Image Processing System Based on a Transputer Tree Network - D. W. Downing and I. B. Bennett, I.E.E. Colloquium on Transputers for Image Processing Applications, London, February 1989, Digest No. 1989/22

[135] Communicating Sequential Processes - C. A. R. Hoare, Communications of the A.C.M., V21, N8 (Aug 78) pp666 - 677

[136] An Image Processing System with Optimising Performance - D. W. Downing and I. B. Bennett - Proceedings of the Occam User Group Sixth Technical Conference, Surrey University, Guildford, 15 April 1987

[137] Performance Maximisation - P. Atkin, Inmos Technical Note 17, 1987

[138] The Pursuit of Deadlock Freedom - A. W. Roscoe and C. A. R. Hoare, P.R.G. Publication, Oxford University, 1986

[139] A. Rosenfeld and A. C. Kak, Digital Picture Processing Second Edition., New York Academic Press, 1982

[140] An Automatic Thresholding Algorithm and its Performance - J. Kittler et al, Proceedings of Seventh International Conference on Pattern Recognition, Montreal, Canada, July 1984

[141] Writing Efficient Programs - J. L. Bentley, Prentice-Hall, U.S.A., 1982

[142] Computer Processing of Line-Drawing Images, H. Freeman, Computing Surveys Vol. 6 No. 1 (March 1974)

[143] Computational Geometry and VLSI - F. Dehne, Proceedings International Conference on VLSI and Computers, Hamburg, May 1987

[144] Two Methods for Finding Convex Hulls of Planar Figures - B. G. Batchelor, Cybernetics and Systems, Vol 11, pp105-113, 1980

[145] Convex Hull of a Chain Coded BLOB - G.R. Wilson, (To be published in I.E.E. Proceedings, Pt E)

[146] Multi-transputer Based Parallel Implementation of Feature Extraction for Object Recognition - D. W. Downing and I. B. Bennett, Proceedings of the Occam User Group Eighth Technical Conference, Sheffield City Polytechnic, Sheffield, U.K. 29 March 1988

[147] Private Correspondence with G.R. Wilson, Gwent College of Higher Education, 1988

[148] (Ed.) B.G. Batchelor, Pattern Recognition, Ideas in Practice, Plenum Press, 1978

[149] R. A. Duda and P. E. Hart, Pattern Classification and Scene Analysis, Wiley 1973

[150] (Ed) S. Levialdi, Digital Image Analysis, Pitman, 1984

[151] (Ed) A. Pugh, Robot Vision, IFS Publications Ltd., 1983

[152] Vision and the Oxford Autonomous Guided Vehicle - M. Brady et al, I.E.E. Colloquium on Computer Vision for Robotics, London, January 1989, Digest No. 1989/9.

[153] Image Processing Transputer Based Machine for Mobile Robot Road Following - R. A. Lotufo et al, I.E.E. Colloquium on Transputers for Image Processing Applications, London, February 1989, Digest No. 1989/22

[154] Dedicated Microprocessor Controller for Hydraulic Robotic Arm - I. B. Bennett, Final year degree project report, Portsmouth Polytechnic, July 1984

[155] Technical Reference Manual for Transputer Development System Z80 Interface - I. B. Bennett and D. W. Downing, Research Report No. 86/IB/1, Gwent College of Higher Education, May 1986